· 全面讲解 Kibana 布局，面板操作说明，仪表盘定制方法
· 支持新版 Elasticsearch 聚合函数功能
· 完整的认证授权体系



三斗室 著

Kibana 中文指南

NOT′REALLY® *A committee - and look at the mess we ended up with*

目錄

前言

Elastic Stack 是原 ELK Stack 在 5.0 版本加入 Beats 套件后的新称呼。

Elastic Stack 在最近两年迅速崛起，成为机器数据分析，或者说实时日志处理领域，开源界的第一选择。和传统的日志处理方案相比，Elastic Stack 具有如下几个优点：

- 处理方式灵活。Elasticsearch 是实时全文索引，不需要像 storm 那样预先编程才能使用；
- 配置简易上手。Elasticsearch 全部采用 JSON 接口，Logstash 是 Ruby DSL 设计，都是目前业界最通用的配置语法设计；
- 检索性能高效。虽然每次查询都是实时计算，但是优秀的设计和实现基本可以达到全天数据查询的秒级响应；
- 集群线性扩展。不管是 Elasticsearch 集群还是 Logstash 集群都是可以线性扩展的；
- 前端操作炫丽。Kibana 界面上，只需要点击鼠标，就可以完成搜索、聚合功能，生成炫丽的仪表板。

当然，Elastic Stack 也并不是实时数据分析界的灵丹妙药。在不恰当的场景，反而会事倍功半。我自 2014 年初开 QQ 群交流 Elastic Stack，发现网友们对 Elastic Stack 的原理概念，常有误解误用；对实现的效果，又多有不能理解或者过多期望而失望之处。更令我惊奇的是，网友们广泛分布在传统企业和互联网公司、开发和运维领域、Linux 和 Windows 平台，大家对非专精领域的知识，一般都缺乏了解，这也成为使用 Elastic Stack 时的一个障碍。

为此，写一本 Elastic Stack 技术指南，帮助大家厘清技术细节，分享一些实战案例，成为我近半年一大心愿。本书大体完工之后，幸得机械工业出版社华章公司青睐，以《ELK Stack 权威指南》之名重修完善并出版，有意收藏者欢迎购买。

本人于 Elastic Stack，虽然接触较早，但本身专于 web 和 app 应用数据方面，动笔以来，得到诸多朋友的帮助，详细贡献名单见合作名单。此外，还要特别感谢曾勇(medcl)同学，完成 ES 在国内的启蒙式分享，并主办 ES 中国用户大会；吴晓刚(wood)同学，积极帮助新用户们，并最早分享了携程的 Elastic Stack 日亿级规模的实例。

欢迎加入 Elastic Stack 交流 QQ 群：315428175。

欢迎捐赠，作者支付宝账号：*rao.chenlin@gmail.com*



Version

2016-10-27 发布了 Elastic Stack 5.0 版。由于变动较大，本书 Git 仓库将 master 分支统一调整为基于 5.0 的状态。

想要查阅过去 k3、k4、logstash-2.x 等不同老版本资料的读者，请下载 ELK release：https://github.com/chenryn/ELKstack-guide-cn/releases/tag/ELK

TODO

限于个人经验、时间和场景，有部分 Elastic Stack 社区比较常见的用法介绍未完成，期待各位同好出手。罗列如下：

- es-hadoop 用例
- beats 开发
- codec/netflow 的详解
- filter/elapsed 的用例
- zeppelin 的 es 用例
- kibana的filter交互用法
- painless的date对象用法：同比环比图
- significant_text aggs用例
- cat nodeattrs接口
- timelion保存成panel的用法
- regionmap用法

- Time Series Visual Builder用法
- Viewing Document Context用法
- Dead Letter Queues讲解

第一部分 Logstash

Logstash is a tool for managing events and logs. You can use it to collect logs, parse them, and store them for later use (like, for searching). -- http://logstash.net

Logstash 项目诞生于 2009 年 8 月 2 日。其作者是世界著名的运维工程师乔丹西塞(JordanSissel)，乔丹西塞当时是著名虚拟主机托管商 DreamHost 的员工，还发布过非常棒的软件打包工具 fpm，并主办着一年一度的 sysadmin advent calendar(advent calendar 文化源自基督教氛围浓厚的 Perl 社区，在每年圣诞来临的 12 月举办，从 12 月 1 日起至 12 月 24 日止，每天发布一篇小短文介绍主题相关技术)。

小贴士：*Logstash* 动手很早，对比一下，*scribed* 诞生于 *2008* 年，*flume* 诞生于 *2010* 年，*Graylog2* 诞生于 *2010* 年，*Fluentd* 诞生于 *2011* 年。

scribed 在 2011 年进入半死不活的状态，大大激发了其他各种开源日志收集处理框架的蓬勃发展，Logstash 也从 2011 年开始进入 commit 密集期并延续至今。

作为一个系出名门的产品，Logstash 的身影多次出现在 Sysadmin Weekly 上，它和它的小伙伴们 Elasticsearch、Kibana 直接成为了和商业产品 Splunk 做比较的开源项目(乔丹西塞曾经在博客上承认设计想法来自 AWS 平台上最大的第三方日志服务商 Loggly，而 Loggly 两位创始人都曾是 Splunk 员工)。

2013 年，Logstash 被 Elasticsearch 公司收购，ELK Stack 正式成为官方用语(随着 beats 的加入改名为 Elastic Stack)。Elasticsearch 本身 也是近两年最受关注的大数据项目之一，三次融资已经超过一亿美元。在 Elasticsearch 开发人员的共同努力下，Logstash 的发布机制，插件架构也愈发科学和合理。

社区文化

日志收集处理框架这么多，像 scribe 是 facebook 出品，flume 是 apache 基金会项目，都算声名赫赫。但 logstash 因乔丹西塞的个人性格，形成了一套独特的社区文化。每一个在 google groups 的 logstash-users 组里问答的人都会看到这么一句话：

Remember: if a new user has a bad time, it's a bug in logstash.

所以，logstash 是一个开放的，极其互助和友好的大家庭。有任何问题，尽管在 github issue，Google groups，Freenode#logstash channel 上发问就好！

基础知识

什么是 *Logstash*？为什么要用 *Logstash*？怎么用 *Logstash*？

本章正是来回答这个问题，或许不完整，但是足够讲述一些基础概念。跟着我们安装章节一步步来，你就可以成功的运行起来自己的第一个 logstash 了。

我可能不会立刻来展示 logstash 配置细节或者运用场景。我认为基础原理和语法的介绍应该更加重要，这些知识未来对你的帮助绝对更大！

所以，认真阅读他们吧！

安装

下载

直接下载官方发布的二进制包的，可以访问 https://www.elastic.co/downloads/logstash 页面找对应操作系统和版本，点击下载即可。不过更推荐使用软件仓库完成安装。

安装

如果你必须得在一些很老的操作系统上运行 Logstash，那你只能用源代码包部署了，记住要自己提前安装好 Java：

```
yum install java-1.8.0-openjdk
export JAVA_HOME=/usr/java
```

软件仓库的配置，主要两大平台如下：

Debian 平台

```
wget -O - http://packages.elasticsearch.org/GPG-KEY-elasticsearc
h | apt-key add -
cat >> /etc/apt/sources.list <<EOF
deb http://packages.elasticsearch.org/logstash/5.0/debian stable
 main
EOF
apt-get update
apt-get install logstash
```

Redhat 平台

```
rpm --import http://packages.elasticsearch.org/GPG-KEY-elasticse
arch
cat > /etc/yum.repos.d/logstash.repo <<EOF
[logstash-5.0]
name=logstash repository for 5.0.x packages
baseurl=http://packages.elasticsearch.org/logstash/5.0/centos
gpgcheck=1
gpgkey=http://packages.elasticsearch.org/GPG-KEY-elasticsearch
enabled=1
EOF
yum clean all
yum install logstash
```

Hello World

和绝大多数 IT 技术介绍一样，我们以一个输出 "hello world" 的形式开始我们的 logstash 学习。

运行

在终端中，像下面这样运行命令来启动 Logstash 进程：

```
# bin/logstash -e 'input{stdin{}}output{stdout{codec=>rubydebug}
}'
```

然后你会发现终端在等待你的输入。没问题，敲入 **Hello World**，回车，然后看看会返回什么结果！

结果

```
{
       "message" => "Hello World",
      "@version" => "1",
    "@timestamp" => "2014-08-07T10:30:59.937Z",
          "host" => "raochenlindeMacBook-Air.local",
}
```

没错！你搞定了！这就是全部你要做的。

解释

每位系统管理员都肯定写过很多类似这样的命令： `cat randdata | awk '{print $2}' | sort | uniq -c | tee sortdata` 。这个管道符 `|` 可以算是 Linux 世界最伟大的发明之一(另一个是“一切皆文件”)。

Logstash 就像管道符一样！

你输入(就像命令行的 `cat`)数据，然后处理过滤(就像 `awk` 或者 `uniq` 之类)数据，最后输出(就像 `tee`)到其他地方。

当然实际上，*Logstash* 是用不同的线程来实现这些的。如果你运行 `top` 命令然后按下 `H` 键，你就可以看到下面这样的输出：

```
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COM
MAND
21401 root      16   0 1249m 303m  10m S 18.6  0.2 866:25.46 |wo
rker
21467 root      15   0 1249m 303m  10m S  3.7  0.2 129:25.59 >el
asticsearch.
21468 root      15   0 1249m 303m  10m S  3.7  0.2 128:53.39 >el
asticsearch.
21400 root      15   0 1249m 303m  10m S  2.7  0.2 108:35.80 <fi
le
21403 root      15   0 1249m 303m  10m S  1.3  0.2  49:31.89 >ou
tput
21470 root      15   0 1249m 303m  10m S  1.0  0.2  56:24.24 >el
asticsearch.
```

小贴士：*logstash* 很温馨的给每个线程都取了名字，输入的叫*xx*，过滤的叫*|xx*

数据在线程之间以 事件 的形式流传。不要叫行，因为 logstash 可以处理多行事件。

Logstash 会给事件添加一些额外信息。最重要的就是 **@timestamp**，用来标记事件的发生时间。因为这个字段涉及到 Logstash 的内部流转，所以必须是一个 joda 对象，如果你尝试自己给一个字符串字段重命名为 `@timestamp` 的话，Logstash 会直接报错。所以，请使用 **filters/date** 插件 来管理这个特殊字段。

此外，大多数时候，还可以见到另外几个：

1. **host** 标记事件发生在哪里。
2. **type** 标记事件的唯一类型。
3. **tags** 标记事件的某方面属性。这是一个数组，一个事件可以有多个标签。

你可以随意给事件添加字段或者从事件里删除字段。事实上事件就是一个 Ruby 对象，或者更简单的理解为就是一个哈希也行。

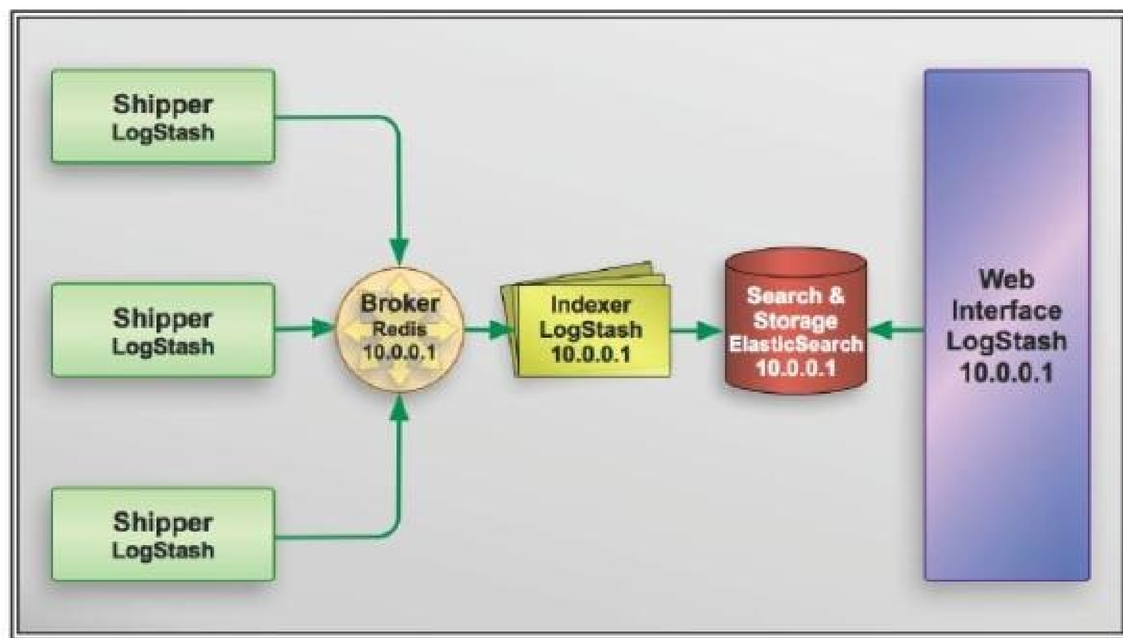
小贴士：每个 *logstash* 过滤插件，都会有四个方法叫 `add_tag`，`remove_tag`，`add_field` 和 `remove_field`。它们在插件过滤匹配成功时生效。

推荐阅读

- 《the life of an event》官网文档
- 《life of a logstash event》Elastic{ON} 上的演讲

配置语法

Logstash 社区通常习惯用 *shipper*，*broker* 和 *indexer* 来描述数据流中不同进程各自的角色。如下图：



不过我见过很多运用场景里都没有用 logstash 作为 *shipper*，或者说没有用 elasticsearch 作为数据存储也就是说也没有 *indexer*。所以，我们其实不需要这些概念。只需要学好怎么使用和配置 logstash 进程，然后把它运用到你的日志管理架构中最合适它的位置就够了。

语法

Logstash 设计了自己的 DSL —— 有点像 Puppet 的 DSL，或许因为都是用 Ruby 语言写的吧 —— 包括有区域，注释，数据类型(布尔值，字符串，数值，数组，哈希)，条件判断，字段引用等。

区段(section)

Logstash 用 `{}` 来定义区域。区域内可以包括插件区域定义，你可以在一个区域内定义多个插件。插件区域内则可以定义键值对设置。示例如下：

```
input {
    stdin {}
    syslog {}
}
```

数据类型

Logstash 支持少量的数据值类型：

- bool

```
debug => true
```

- string

```
host => "hostname"
```

- number

```
port => 514
```

- array

```
match => ["datetime", "UNIX", "ISO8601"]
```

- hash

```
options => {
    key1 => "value1",
    key2 => "value2"
}
```

注意：如果你用的版本低于 1.2.0，哈希的语法跟数组是一样的，像下面这样写：

```
match => [ "field1", "pattern1", "field2", "pattern2" ]
```

字段引用(field reference)

字段是 `Logstash::Event` 对象的属性。我们之前提过事件就像一个哈希一样，所以你可以想象字段就像一个键值对。

小贴士：我们叫它字段，因为 *Elasticsearch* 里是这么叫的。

如果你想在 Logstash 配置中使用字段的值，只需要把字段的名字写在中括号 `[]` 里就行了，这就叫字段引用。

对于 嵌套字段(也就是多维哈希表，或者叫哈希的哈希)，每层的字段名都写在 `[]` 里就可以了。比如，你可以从 geoip 里这样获取 *longitude* 值(是的，这是个笨办法，实际上有单独的字段专门存这个数据的)：

```
[geoip][location][0]
```

小贴士：*logstash* 的数组也支持倒序下标，即 *`[geoip][location][-1]`* 可以获取数组最后一个元素的值。

Logstash 还支持变量内插，在字符串里使用字段引用的方法是这样：

```
"the longitude is %{[geoip][location][0]}"
```

条件判断(condition)

Logstash从 1.3.0 版开始支持条件判断和表达式。

表达式支持下面这些操作符：

- `==` (等于), `!=` (不等于), `<` (小于), `>` (大于), `<=` (小于等于), `>=` (大于等于)
- `=~` (匹配正则), `!~` （不匹配正则）
- `in` (包含), `not in` (不包含)
- `and` (与), `or` (或), nand(非与), xor(非或)
- `()` (复合表达式), `!()` (对复合表达式结果取反)

通常来说，你都会在表达式里用到字段引用。为了尽量展示全面各种表达式，下面虚拟一个示例：

```
if "_grokparsefailure" not in [tags] {
} else if [status] !~ /^2\d\d/ or ( [url] == "/noc.gif" nand [ge
oip][city] != "beijing" ) {
} else {
}
```

命令行参数

Logstash 提供了一个 shell 脚本叫 `logstash` 方便快速运行。它支持以下参数：

- -e

意即执行。我们在 "Hello World" 的时候已经用过这个参数了。事实上你可以不写任何具体配置，直接运行 `bin/logstash -e ''` 达到相同效果。这个参数的默认值是下面这样：

```
input {
    stdin { }
}
output {
    stdout { }
}
```

- --config 或 -f

意即文件。真实运用中，我们会写很长的配置，甚至可能超过 shell 所能支持的 1024 个字符长度。所以我们必把配置固化到文件里，然后通过 `bin/logstash -f agent.conf` 这样的形式来运行。

此外，logstash 还提供一个方便我们规划和书写配置的小功能。你可以直接用 `bin/logstash -f /etc/logstash.d/` 来运行。logstash 会自动读取 `/etc/logstash.d/` 目录下所有 `*.conf` 的文本文件，然后在自己内存里拼接成一个完整的大配置文件，再去执行。

注意：

logstash 列出目录下所有文件时，是字母排序的。而 logstash 配置段的 filter 和 output 都是顺序执行，所以顺序非常重要。采用多文件管理的用户，推荐采用数字编号方式命名配置文件，同时在配置中，严谨采用 `if` 判断限定不同日志的动作。

- --configtest 或 -t

意即测试。用来测试 Logstash 读取到的配置文件语法是否能正常解析。Logstash 配置语法是用 grammar.treetop 定义的。尤其是使用了上一条提到的读取目录方式的读者，尤其要提前测试。

- --log 或 -l

意即日志。Logstash 默认输出日志到标准错误。生产环境下你可以通过 `bin/logstash -l logs/logstash.log` 命令来统一存储日志。

- --pipeline-workers 或 -w

运行 filter 和 output 的 pipeline 线程数量。默认是 CPU 核数。

- --pipeline-batch-size 或 -b

每个 Logstash pipeline 线程，在执行具体的 filter 和 output 函数之前，最多能累积的日志条数。默认是 125 条。越大性能越好，同样也会消耗越多的 JVM 内存。

- --pipeline-batch-delay 或 -u

每个 Logstash pipeline 线程，在打包批量日志的时候，最多等待几毫秒。默认是 5 ms。

- --pluginpath 或 -P

可以写自己的插件，然后用 `bin/logstash --pluginpath /path/to/own/plugins` 加载它们。

- --verbose

输出一定的调试日志。

- --debug

输出更多的调试日志。

设置文件

从 Logstash 5.0 开始，新增了 `$LS_HOME/config/logstash.yml` 文件，可以将所有的命令行参数都通过 YAML 文件方式设置。同时为了反映命令行配置参数的层级关系，参数也都改成用.而不是-了。

```
pipeline:
    workers: 24
    batch:
        size: 125
        delay: 5
```

plugin的安装

从 logstash 1.5.0 版本开始，logstash 将所有的插件都独立拆分成 gem 包。这样，每个插件都可以独立更新，不用等待 logstash 自身做整体更新的时候才能使用了。

为了达到这个目标，logstash 配置了专门的 plugins 管理命令。

plugin 用法说明

```
Usage:
    bin/logstash-plugin [OPTIONS] SUBCOMMAND [ARG] ...

Parameters:
    SUBCOMMAND                    subcommand
    [ARG] ...                     subcommand arguments

Subcommands:
    install                       Install a plugin
    uninstall                     Uninstall a plugin
    update                        Install a plugin
    list                          List all installed plugins

Options:
    -h, --help                    print help
```

示例

首先，你可以通过 `bin/logstash-plugin list` 查看本机现在有多少插件可用。(其实就在 vendor/bundle/jruby/1.9/gems/ 目录下)

然后，假如你看到 `https://github.com/logstash-plugins/` 下新发布了一个 `logstash-output-webhdfs` 模块(当然目前还没有)。打算试试，就只需要运行：

```
bin/logstash-plugin install logstash-output-webhdfs
```

就可以了。

同样，假如是升级，只需要运行：

```
bin/logstash-plugin update logstash-input-tcp
```

即可。

本地插件安装

`bin/logstash-plugin` 不单可以通过 rubygems 平台安装插件，还可以读取本地路径的 gem 文件。这对自定义插件或者无外接网络的环境都非常有效：

```
bin/logstash-plugin install /path/to/logstash-filter-crash.gem
```

执行成功以后。你会发现，logstash-5.0.0 目录下的 Gemfile 文件最后会多出一段内容：

```
gem "logstash-filter-crash", "1.1.0", :path => "vendor/local_gem
s/d354312c/logstash-filter-mweibocrash-1.1.0"
```

同时 Gemfile.jruby-1.9.lock 文件开头也会多出一段内容：

```
PATH
  remote: vendor/local_gems/d354312c/logstash-filter-crash-1.1.0
  specs:
    logstash-filter-crash (1.1.0)
      logstash-core (>= 1.4.0, < 2.0.0)
```

长期运行

完成上一节的初次运行后，你肯定会发现一点：一旦你按下 Ctrl+C，停下标准输入输出，logstash 进程也就随之停止了。作为一个肯定要长期运行的程序，应该怎么处理呢？

本章节问题对于一个运维来说应该属于基础知识，鉴于 *ELK* 用户很多其实不是运维，添加这段内容。

办法有很多种，下面介绍四种最常用的办法：

标准的 service 方式

采用 RPM、DEB 发行包安装的读者，推荐采用这种方式。发行包内，都自带有 sysV 或者 systemd 风格的启动程序/配置，你只需要直接使用即可。

以 RPM 为例， `/etc/init.d/logstash` 脚本中，会加载 `/etc/init.d/functions` 库文件，利用其中的 `daemon` 函数，将 logstash 进程作为后台程序运行。

所以，你只需把自己写好的配置文件，统一放在 `/etc/logstash/conf.d` 目录下(注意目录下所有配置文件都应该是 **.conf** 结尾，且不能有其他文本文件存在。因为 logstash agent 启动的时候是读取全文件夹的)，然后运行 `service logstash start` 命令即可。

最基础的 nohup 方式

这是最简单的方式，也是 linux 新手们很容易搞混淆的一个经典问题：

```
command
command > /dev/null
command > /dev/null 2>&1
command &
command > /dev/null &
command > /dev/null 2>&1 &
command &> /dev/null
nohup command &> /dev/null
```

请回答以上命令的异同......

具体不一一解释了。直接说答案，想要维持一个长期后台运行的 logstash，你需要同时在命令前面加 `nohup`，后面加 `&`。

更优雅的 SCREEN 方式

screen 算是 linux 运维一个中高级技巧。通过 screen 命令创建的环境下运行的终端命令，其父进程不是 sshd 登录会话，而是 screen 。这样就可以即避免用户退出进程消失的问题，又随时能重新接管回终端继续操作。

创建独立的 screen 命令如下：

```
screen -dmS elkscreen_1
```

接管连入创建的 `elkscreen_1` 命令如下：

```
screen -r elkscreen_1
```

然后你可以看到一个一模一样的终端，运行 logstash 之后，不要按 Ctrl+C，而是按 Ctrl+A+D 键，断开环境。想重新接管，依然 `screen -r elkscreen_1` 即可。

如果创建了多个 screen，查看列表命令如下：

```
screen -list
```

最推荐的 daemontools 方式

不管是 nohup 还是 screen，都不是可以很方便管理的方式，在运维管理一个 ELK 集群的时候，必须寻找一种尽可能简洁的办法。所以，对于需要长期后台运行的大量程序(注意大量，如果就一个进程，还是学习一下怎么写 init 脚本吧)，推荐大家使用一款 daemontools 工具。

daemontools 是一个软件名称，不过配置略复杂。所以这里我其实是用其名称来指代整个同类产品，包括但不限于 python 实现的 supervisord，perl 实现的 ubic，ruby 实现的 god 等。

以 supervisord 为例，因为这个出来的比较早，可以直接通过 EPEL 仓库安装。

```
yum -y install supervisord --enablerepo=epel
```

在 `/etc/supervisord.conf` 配置文件里添加内容，定义你要启动的程序：

```
[program:elkpro_1]
environment=LS_HEAP_SIZE=5000m
directory=/opt/logstash
command=/opt/logstash/bin/logstash -f /etc/logstash/pro1.conf -w
 10 -l /var/log/logstash/pro1.log
[program:elkpro_2]
environment=LS_HEAP_SIZE=5000m
directory=/opt/logstash
command=/opt/logstash/bin/logstash -f /etc/logstash/pro2.conf -w
 10 -l /var/log/logstash/pro2.log
```

然后启动 `service supervisord start` 即可。

logstash 会以 supervisord 子进程的身份运行，你还可以使用 `supervisorctl` 命令，单独控制一系列 logstash 子进程中某一个进程的启停操作：

`supervisorctl stop elkpro_2`

输入插件(Input)

在 "Hello World" 示例中，我们已经见到并介绍了 logstash 的运行流程和配置的基础语法。从这章开始，我们就要逐一介绍 logstash 流程中比较常用的一些插件，并在介绍中针对其主要适用的场景，推荐的配置，作一些说明。

限于篇幅，接下来内容中，配置示例不一定能贴完整。请记住一个原则：Logstash 配置一定要有一个 input 和一个 output。在演示过程中，如果没有写明 input，默认就会使用 "hello world" 里我们已经演示过的 *input/stdin* ，同理，没有写明的 output 就是 *output/stdout*。

以上请读者自明。

读取文件(File)

分析网站访问日志应该是一个运维工程师最常见的工作了。所以我们先学习一下怎么用 logstash 来处理日志文件。

Logstash 使用一个名叫 *FileWatch* 的 Ruby Gem 库来监听文件变化。这个库支持 glob 展开文件路径，而且会记录一个叫 *.sincedb* 的数据库文件来跟踪被监听的日志文件的当前读取位置。所以，不要担心 logstash 会漏过你的数据。

sincedb 文件中记录了每个被监听的文件的 *inode, major number, minor number* 和 *pos*。

配置示例

```
input {
    file {
        path => ["/var/log/*.log", "/var/log/message"]
        type => "system"
        start_position => "beginning"
    }
}
```

解释

有一些比较有用的配置项，可以用来指定 *FileWatch* 库的行为：

- discover_interval

logstash 每隔多久去检查一次被监听的 `path` 下是否有新文件。默认值是 15 秒。

- exclude

不想被监听的文件可以排除出去，这里跟 `path` 一样支持 glob 展开。

- close_older

一个已经监听中的文件，如果超过这个值的时间内没有更新内容，就关闭监听它的文件句柄。默认是 3600 秒，即一小时。

- ignore_older

在每次检查文件列表的时候，如果一个文件的最后修改时间超过这个值，就忽略这个文件。默认是 86400 秒，即一天。

- sincedb_path

如果你不想用默认的 `$HOME/.sincedb` (Windows 平台上在 `C:\Windows\System32\config\systemprofile\.sincedb`)，可以通过这个配置定义 sincedb 文件到其他位置。

- sincedb_write_interval

logstash 每隔多久写一次 sincedb 文件，默认是 15 秒。

- stat_interval

logstash 每隔多久检查一次被监听文件状态（是否有更新），默认是 1 秒。

- start_position

logstash 从什么位置开始读取文件数据，默认是结束位置，也就是说 logstash 进程会以类似 `tail -F` 的形式运行。如果你是要导入原有数据，把这个设定改成 "beginning"，logstash 进程就从头开始读取，类似 `less +F` 的形式运行。

注意

1. 通常你要导入原有数据进 Elasticsearch 的话，你还需要 filter/date 插件来修改默认的"@timestamp" 字段值。稍后会学习这方面的知识。
2. *FileWatch* 只支持文件的绝对路径，而且会不自动递归目录。所以有需要的话，请用数组方式都写明具体哪些文件。
3. *LogStash::Inputs::File* 只是在进程运行的注册阶段初始化一个 *FileWatch* 对象。所以它不能支持类似 fluentd 那样的 `path => "/path/to/%{+yyyy/MM/dd/hh}.log"` 写法。达到相同目的，你只能写成 `path => "/path/to/*/*/*/*.log"` 。FileWatch 模块提供了一个稍微简单一点的写法： `/path/to/**/*.log` ，用 `**` 来缩写表示递归全部子目录。
4. 在单个 input/file 中监听的文件数量太多的话，每次启动扫描构建监听队列会消

耗较多的时间。给使用者的感觉好像读取不到一样，这是正常现象。

5. `start_position` 仅在该文件从未被监听过的时候起作用。如果 sincedb 文件中已经有这个文件的 inode 记录了，那么 logstash 依然会从记录过的 pos 开始读取数据。所以重复测试的时候每回需要删除 sincedb 文件(官方博客上提供了另一个巧妙的思路：将 `sincedb_path` 定义为 `/dev/null` ，则每次重启自动从头开始读)。
6. 因为 windows 平台上没有 inode 的概念，Logstash 某些版本在 windows 平台上监听文件不是很靠谱。windows 平台上，推荐考虑使用 nxlog 作为收集端，参阅本书稍后章节。

标准输入(Stdin)

我们已经见过好几个示例使用 `stdin` 了。这也应该是 logstash 里最简单和基础的插件了。

所以，在这段中，我们可以学到一些未来每个插件都会有的一些方法。

配置示例

```
input {
    stdin {
        add_field => {"key" => "value"}
        codec => "plain"
        tags => ["add"]
        type => "std"
    }
}
```

运行结果

用上面的新 `stdin` 设置重新运行一次最开始的 hello world 示例。我建议大家把整段配置都写入一个文本文件，然后运行命令： `bin/logstash -f stdin.conf` 。输入 "hello world" 并回车后，你会在终端看到如下输出：

```
{
       "message" => "hello world",
      "@version" => "1",
    "@timestamp" => "2014-08-08T06:48:47.789Z",
          "type" => "std",
          "tags" => [
        [0] "add"
    ],
           "key" => "value",
          "host" => "raochenlindeMacBook-Air.local"
}
```

解释

type 和 *tags* 是 logstash 事件中两个特殊的字段。通常来说我们会在输入区段中通过 *type* 来标记事件类型 —— 我们肯定是提前能知道这个事件属于什么类型的。而 *tags* 则是在数据处理过程中，由具体的插件来添加或者删除的。

最常见的用法是像下面这样：

```
input {
    stdin {
        type => "web"
    }
}
filter {
    if [type] == "web" {
        grok {
            match => ["message", %{COMBINEDAPACHELOG}]
        }
    }
}
output {
    if "_grokparsefailure" in [tags] {
        nagios_nsca {
            nagios_status => "1"
        }
    } else {
        elasticsearch {
        }
    }
}
```

看起来蛮复杂的，对吧？

继续学习，你也可以写出来的。

读取 Syslog 数据

syslog 可能是运维领域最流行的数据传输协议了。当你想从设备上收集系统日志的时候，syslog 应该会是你的第一选择。尤其是网络设备，比如思科 —— syslog 几乎是唯一可行的办法。

我们这里不解释如何配置你的 `syslog.conf` , `rsyslog.conf` 或者 `syslog-ng.conf` 来发送数据，而只讲如何把 logstash 配置成一个 syslog 服务器来接收数据。

有关 `rsyslog` 的用法，稍后的类型项目一节中，会有更详细的介绍。

配置示例

```
input {
  syslog {
    port => "514"
  }
}
```

运行结果

作为最简单的测试，我们先暂停一下本机的 `syslogd` (或 `rsyslogd`)进程，然后启动 logstash 进程（这样就不会有端口冲突问题）。现在，本机的 syslog 就会默认发送到 logstash 里了。我们可以用自带的 `logger` 命令行工具发送一条 "Hello World"信息到 syslog 里（即 logstash 里）。看到的 logstash 输出像下面这样：

```
{
               "message" => "Hello World",
              "@version" => "1",
            "@timestamp" => "2014-08-08T09:01:15.911Z",
                  "host" => "127.0.0.1",
              "priority" => 31,
             "timestamp" => "Aug  8 17:01:15",
             "logsource" => "raochenlindeMacBook-Air.local",
               "program" => "com.apple.metadata.mdflagwriter",
                   "pid" => "381",
              "severity" => 7,
              "facility" => 3,
        "facility_label" => "system",
        "severity_label" => "Debug"
}
```

解释

Logstash 是用 `UDPSocket`, `TCPServer` 和 `LogStash::Filters::Grok` 来实现 `LogStash::Inputs::Syslog` 的。所以你其实可以直接用 logstash 配置实现一样的效果：

```
input {
  tcp {
    port => "8514"
  }
}
filter {
  grok {
    match => ["message", "%{SYSLOGLINE}" ]
  }
  syslog_pri { }
}
```

最佳实践

建议在使用 `LogStash::Inputs::Syslog` 的时候走 **TCP** 协议来传输数据。

因为具体实现中，UDP 监听器只用了一个线程，而 TCP 监听器会在接收每个连接的时候都启动新的线程来处理后续步骤。

如果你已经在使用 UDP 监听器收集日志，用下行命令检查你的 UDP 接收队列大小：

```
# netstat -plnu | awk 'NR==1 || $4~/:514$/{print $2}'
Recv-Q
228096
```

228096 是 UDP 接收队列的默认最大大小，这时候 linux 内核开始丢弃数据包了！

强烈建议使用 `LogStash::Inputs::TCP` 和 `LogStash::Filters::Grok` 配合实现同样的 **syslog** 功能！

虽然 LogStash::Inputs::Syslog 在使用 TCPServer 的时候可以采用多线程处理数据的接收，但是在同一个客户端数据的处理中，其 grok 和 date 是一直在该线程中完成的，这会导致总体上的处理性能几何级的下降 —— 经过测试，TCPServer 每秒可以接收 50000 条数据，而在同一线程中启用 grok 后每秒只能处理 5000 条，再加上 date 只能达到 500 条！

才将这两步拆分到 filters 阶段后，logstash 支持对该阶段插件单独设置多线程运行，大大提高了总体处理性能。在相同环境下， `logstash -f tcp.conf -w 20` 的测试中，总体处理性能可以达到每秒 30000 条数据！

注：测试采用 *logstash* 作者提供的 *`yes "<44>May 19 18:30:17 snack jls: foo bar 32" | nc localhost 3000`* 命令。出处见：*https://github.com/jordansissel/experiments/blob/master/ruby/jruby-netty/syslog-server/Makefile*

小贴士

如果你实在没法切换到 TCP 协议，你可以自己写程序，或者使用其他基于异步 IO 框架(比如 libev)的项目。下面是一个简单的异步 IO 实现 UDP 监听数据输入 Elasticsearch 的示例：

https://gist.github.com/chenryn/7c922ac424324ee0d695

读取网络数据(TCP)

未来你可能会用 Redis 服务器或者其他的消息队列系统来作为 logstash broker 的角色。不过 Logstash 其实也有自己的 TCP/UDP 插件，在临时任务的时候，也算能用，尤其是测试环境。

小贴士：虽然 `LogStash::Inputs::TCP` 用 *Ruby* 的 `Socket` 和 `OpenSSL` 库实现了高级的 *SSL* 功能，但 *Logstash* 本身只能在 `SizedQueue` 中缓存 *20* 个事件。这就是我们建议在生产环境中换用其他消息队列的原因。

配置示例

```
input {
    tcp {
        port => 8888
        mode => "server"
        ssl_enable => false
    }
}
```

常见场景

目前来看，`LogStash::Inputs::TCP` 最常见的用法就是配合 `nc` 命令导入旧数据。在启动 logstash 进程后，在另一个终端运行如下命令即可导入数据：

```
# nc 127.0.0.1 8888 < olddata
```

这种做法比用 `LogStash::Inputs::File` 好，因为当 nc 命令结束，我们就知道数据导入完毕了。而用 input/file 方式，logstash 进程还会一直等待新数据输入被监听的文件，不能直接看出是否任务完成了。

编码插件(Codec)

Codec 是 logstash 从 1.3.0 版开始新引入的概念(*Codec* 来自 *Co*der/*dec*oder 两个单词的首字母缩写)。

在此之前，logstash 只支持纯文本形式输入，然后以过滤器处理它。但现在，我们可以在输入 期处理不同类型的数据，这全是因为有了 **codec** 设置。

所以，这里需要纠正之前的一个概念。Logstash 不只是一个 `input | filter | output` 的数据流，而是一个 `input | decode | filter | encode | output` 的数据流！*codec* 就是用来 decode、encode 事件的。

codec 的引入，使得 logstash 可以更好更方便的与其他有自定义数据格式的运维产品共存，比如 graphite、fluent、netflow、collectd，以及使用 msgpack、json、edn 等通用数据格式的其他产品等。

事实上，我们在第一个 "hello world" 用例中就已经用过 *codec* 了 —— *rubydebug* 就是一种 *codec*！虽然它一般只会用在 stdout 插件中，作为配置测试或者调试的工具。

小贴士：这个五段式的流程说明源自 *Perl* 版的 *Logstash (*后来改名叫 *Message::Passing* 模块*)*的设计。本书最后会对该模块稍作介绍。

采用 JSON 编码

在早期的版本中，有一种降低 logstash 过滤器的 CPU 负载消耗的做法盛行于社区(在当时的 cookbook 上有专门的一节介绍)：直接输入预定义好的 **JSON** 数据，这样就可以省略掉 **filter/grok** 配置！

这个建议依然有效，不过在当前版本中需要稍微做一点配置变动 —— 因为现在有专门的 *codec* 设置。

配置示例

社区常见的示例都是用的 Apache 的 customlog。不过我觉得 Nginx 是一个比 Apache 更常用的新型 web 服务器，所以我这里会用 nginx.conf 做示例：

```
logformat json '{"@timestamp":"$time_iso8601",'
               '"@version":"1",'
               '"host":"$server_addr",'
               '"client":"$remote_addr",'
               '"size":$body_bytes_sent,'
               '"responsetime":$request_time,'
               '"domain":"$host",'
               '"url":"$uri",'
               '"status":"$status"}';
access_log /var/log/nginx/access.log_json json;
```

注意：在 `$request_time` 和 `$body_bytes_sent` 变量两头没有双引号 `"`，这两个数据在 *JSON* 里应该是数值类型！

重启 nginx 应用，然后修改你的 input/file 区段配置成下面这样：

```
input {
    file {
        path => "/var/log/nginx/access.log_json"
        codec => "json"
    }
}
```

运行结果

下面访问一下你 nginx 发布的 web 页面，然后你会看到 logstash 进程输出类似下面这样的内容：

```
{
      "@timestamp" => "2014-03-21T18:52:25.000+08:00",
        "@version" => "1",
            "host" => "raochenlindeMacBook-Air.local",
          "client" => "123.125.74.53",
            "size" => 8096,
    "responsetime" => 0.04,
          "domain" => "www.domain.com",
             "url" => "/path/to/file.suffix",
          "status" => "200"
}
```

小贴士

对于一个 web 服务器的访问日志，看起来已经可以很好的工作了。不过如果 Nginx 是作为一个代理服务器运行的话，访问日志里有些变量，比如说 `$upstream_response_time`，可能不会一直是数字，它也可能是一个 `"-"` 字符串！这会直接导致 logstash 对输入数据验证报异常。

有两个办法解决这个问题：

1. 用 `sed` 在输入之前先替换 `-` 成 `0`。

运行 logstash 进程时不再读取文件而是标准输入，这样命令就成了下面这个样子：

```
tail -F /var/log/nginx/proxy_access.log_json \
    | sed 's/upstreamtime":-/upstreamtime":0/' \
    | /usr/local/logstash/bin/logstash -f /usr/local/logstash/et
c/proxylog.conf
```

1. 日志格式中统一记录为字符串格式(即都带上双引号 `"`)，然后再在 logstash 中用 `filter/mutate` 插件来变更应该是数值类型的字符字段的值类型。

有关 `LogStash::Filters::Mutate` 的内容，本书稍后会有介绍。

合并多行数据(Multiline)

有些时候，应用程序调试日志会包含非常丰富的内容，为一个事件打印出很多行内容。这种日志通常都很难通过命令行解析的方式做分析。

而 logstash 正为此准备好了 *codec/multiline* 插件！

小贴士：*multiline* 插件也可以用于其他类似的堆栈式信息，比如 *linux* 的内核日志。

配置示例

```
input {
    stdin {
        codec => multiline {
            pattern => "^\["
            negate => true
            what => "previous"
        }
    }
}
```

运行结果

运行 logstash 进程，然后在等待输入的终端中输入如下几行数据：

```
[Aug/08/08 14:54:03] hello world
[Aug/08/09 14:54:04] hello logstash
    hello best practice
    hello raochenlin
[Aug/08/10 14:54:05] the end
```

你会发现 logstash 输出下面这样的返回：

```
{
    "@timestamp" => "2014-08-09T13:32:03.368Z",
       "message" => "[Aug/08/08 14:54:03] hello world\n",
      "@version" => "1",
          "host" => "raochenlindeMacBook-Air.local"
}
{
    "@timestamp" => "2014-08-09T13:32:24.359Z",
       "message" => "[Aug/08/09 14:54:04] hello logstash\n\n
hello best practice\n\n    hello raochenlin\n",
      "@version" => "1",
          "tags" => [
        [0] "multiline"
    ],
          "host" => "raochenlindeMacBook-Air.local"
}
```

你看，后面这个事件，在 "message" 字段里存储了三行数据！

小贴士：你可能注意到输出的事件中都没有最后的*"the end"*字符串。这是因为你最后输入的回车符 `\n` 并不匹配设定的 `^\[` 正则表达式，*logstash* 还得等下一行数据直到匹配成功后才会输出这个事件。

解释

其实这个插件的原理很简单，就是把当前行的数据添加到前面一行后面，，直到新进的当前行匹配 `^\[` 正则为止。

这个正则还可以用 grok 表达式，稍后你就会学习这方面的内容。

Log4J 的另一种方案

说到应用程序日志，log4j 肯定是第一个被大家想到的。使用 `codec/multiline` 也确实是一个办法。

不过，如果你本身就是开发人员，或者可以推动程序修改变更的话，logstash 还提供了另一种处理 log4j 的方式：input/log4j。与 `codec/multiline` 不同，这个插件是直接调用了 `org.apache.log4j.spi.LoggingEvent` 处理 TCP 端口接收的数据。稍后章节会详细讲述 log4j 的用法。

推荐阅读

https://github.com/logstash-plugins/logstash-patterns-core/blob/master/patterns/java

collectd简述

本节作者：*crazw*

collectd 是一个守护(daemon)进程，用来收集系统性能和提供各种存储方式来存储不同值的机制。它会在系统运行和存储信息时周期性的统计系统的相关统计信息。利用这些信息有助于查找当前系统性能瓶颈（如作为性能分析 `performance analysis` ）和预测系统未来的 load（如能力部署 `capacity planning` ）等

下面简单介绍一下: collectd的部署以及与logstash对接的相关配置实例

collectd的安装

软件仓库安装(推荐)

collectd官方有一个的软件仓库: https://pkg.ci.collectd.org ，构建有 `RHEL/CentOS` (rpm)， `Debian/Ubuntu` (deb)的软件包，如果你使用的操作系统属于上述，那么推荐使用软件仓库安装。

目前collectd官方维护3个版本: `5.4` , `5.5` , `5.6` 。根据需要选择合适的版本。

Debian/Ubuntu仓库安装(示例中使用 `5.5` 版本):

```
echo "deb http://pkg.ci.collectd.org/deb $(lsb_release -sc) collectd-5.5" | sudo tee /etc/apt/sources.list.d/collectd.list
curl -s https://pkg.ci.collectd.org/pubkey.asc | sudo apt-key add -
sudo apt-get update && sudo apt-get install -y collectd
```

> **NOTE**: Debian/Ubuntu软件仓库自带有 `collectd` 软件包，如果软件仓库自带的版本足够你使用，那么可以不用添加仓库，直接通过 `apt-get install collectd` 即可。

RHEL/CentOS仓库安装(示例中使用 `5.5` 版本):

```
cat > /etc/yum.repos.d/collectd.repo <<EOF
[collectd-5.5]
name=collectd-5.5
baseurl=http://pkg.ci.collectd.org/rpm/collectd-5.5/epel-\$releasever-\$basearch/
gpgcheck=1
gpgkey=http://pkg.ci.collectd.org/pubkey.asc
EOF

yum install -y collectd
# 其他collectd插件需要安装对应的collectd-xxxx软件包
```

源码安装collectd

```
# collectd目前维护3个版本, 5.4, 5.5, 5.6。根据自己需要选择版本
wget http://collectd.org/files/collectd-5.4.1.tar.gz
tar zxvf collectd-5.4.1.tar.gz
cd collectd-5.4.1
./configure --prefix=/usr --sysconfdir=/etc --localstatedir=/var
 --libdir=/usr/lib --mandir=/usr/share/man --enable-all-plugins
make && make install
```

解决依赖(RH系列):

```
rpm -ivh "http://dl.fedoraproject.org/pub/epel/6/i386/epel-release-6-8.noarch.rpm"
yum -y install libcurl libcurl-devel rrdtool rrdtool-devel perl-rrdtool rrdtool-perl libgcrypt-devel gcc make gcc-c++ liboping liboping-devel perl-CPAN net-snmp net-snmp-devel
```

安装启动脚本

```
cp contrib/redhat/init.d-collectd /etc/init.d/collectd
chmod +x /etc/init.d/collectd
```

启动collectd

```
service collectd start
```

collectd的配置

以下配置可以实现对服务器基本的**CPU**、内存、网卡流量、磁盘 **IO** 以及磁盘空间占用情况的监控:

```
Hostname "host.example.com"
LoadPlugin interface
LoadPlugin cpu
LoadPlugin memory
LoadPlugin network
LoadPlugin df
LoadPlugin disk
<Plugin interface>
    Interface "eth0"
    IgnoreSelected false
</Plugin>
<Plugin network>
    # logstash 的 IP 地址和 collectd 的数据接收端口号>
    # 如果logstash和collectd在同一台主机上也可以用环回地址127.0.0.1
    <Server "10.0.0.1" "25826">
    </Server>
</Plugin>
```

logstash的配置

以下配置实现通过 logstash 监听 `25826` 端口，接收从 collectd 发送过来的各项检测数据。

logstash默认自带有 `collectd` 的codec插件，详见官方文档:
https://www.elastic.co/guide/en/logstash/current/plugins-codecs-collectd.html

示例：

```
input {
    udp {
        port => 25826
        buffer_size => 1452
        workers => 3          # Default is 2
        queue_size => 30000   # Default is 2000
        codec => collectd { }
        type => "collectd"
    }
}
```

运行结果

下面是简单的一个输出结果：

```
{
  "_index": "logstash-2014.12.11",
  "_type": "collectd",
  "_id": "dS6vVz4aRtK5xS86kwjZnw",
  "_score": null,
  "_source": {
    "host": "host.example.com",
    "@timestamp": "2014-12-11T06:28:52.118Z",
    "plugin": "interface",
    "plugin_instance": "eth0",
    "collectd_type": "if_packets",
    "rx": 19147144,
    "tx": 3608629,
    "@version": "1",
    "type": "collectd",
    "tags": [
      "_grokparsefailure"
    ]
  },
  "sort": [
    1418279332118
  ]
}
```

参考资料

- collectd支持收集的数据类型：http://git.verplant.org/?p=collectd.git;a=blob;hb=master;f=README
- collectd收集各数据类型的配置参考资料：http://collectd.org/documentation/manpages/collectd.conf.5.shtml
- collectd简单配置文件示例：https://gist.github.com/untergeek/ab85cb86a9bf39f1fc6d

netflow

```
input {
    udp {
      port => 9995
      codec => netflow {
        definitions => "/home/administrator/logstash-1.4.2/lib/l
ogstash/codecs/netflow/netflow.yaml"
        versions => [5]
      }
    }
  }

  output {
    stdout { codec => rubydebug }
    if ( [host] =~ "10\.1\.1[12]\.1" ) {
      elasticsearch {
        index => "logstash_netflow5-%{+YYYY.MM.dd}"
        host => "localhost"
      }
    } else {
      elasticsearch {
        index => "logstash-%{+YYYY.MM.dd}"
        host => "localhost"
      }
    }
  }
```

```
curl -XPUT localhost:9200/_template/logstash_netflow5 -d '{
    "template" : "logstash_netflow5-*",
    "settings": {
      "index.refresh_interval": "5s"
    },
    "mappings" : {
      "_default_" : {
        "_all" : {"enabled" : false},
```

```
        "properties" : {
          "@version": { "index": "analyzed", "type": "integer" }
,
          "@timestamp": { "index": "analyzed", "type": "date" },
          "netflow": {
            "dynamic": true,
            "type": "object",
            "properties": {
              "version": { "index": "analyzed", "type": "integer
" },
              "flow_seq_num": { "index": "not_analyzed", "type":
 "long" },
              "engine_type": { "index": "not_analyzed", "type":
"integer" },
              "engine_id": { "index": "not_analyzed", "type": "i
nteger" },
              "sampling_algorithm": { "index": "not_analyzed", "
type": "integer" },
              "sampling_interval": { "index": "not_analyzed", "t
ype": "integer" },
              "flow_records": { "index": "not_analyzed", "type":
 "integer" },
              "ipv4_src_addr": { "index": "analyzed", "type": "i
p" },
              "ipv4_dst_addr": { "index": "analyzed", "type": "i
p" },
              "ipv4_next_hop": { "index": "analyzed", "type": "i
p" },
              "input_snmp": { "index": "not_analyzed", "type": "
long" },
              "output_snmp": { "index": "not_analyzed", "type":
"long" },
              "in_pkts": { "index": "analyzed", "type": "long" }
,
              "in_bytes": { "index": "analyzed", "type": "long"
},
              "first_switched": { "index": "not_analyzed", "type
": "date" },
              "last_switched": { "index": "not_analyzed", "type"
: "date" },
```

```
              "l4_src_port": { "index": "analyzed", "type": "lon
g" },
              "l4_dst_port": { "index": "analyzed", "type": "lon
g" },
              "tcp_flags": { "index": "analyzed", "type": "integ
er" },
              "protocol": { "index": "analyzed", "type": "intege
r" },
              "src_tos": { "index": "analyzed", "type": "integer
" },
              "src_as": { "index": "analyzed", "type": "integer"
 },
              "dst_as": { "index": "analyzed", "type": "integer"
 },
              "src_mask": { "index": "analyzed", "type": "intege
r" },
              "dst_mask": { "index": "analyzed", "type": "intege
r" }
            }
          }
        }
      }
    }
  }'
```

过滤器插件(Filter)

丰富的过滤器插件的存在是 logstash 威力如此强大的重要因素。名为过滤器，其实提供的不单单是过滤的功能。在本章我们就会重点介绍几个插件，它们扩展了进入过滤器的原始数据，进行复杂的逻辑处理，甚至可以无中生有的添加新的 logstash 事件到后续的流程中去！

时间处理(Date)

之前章节已经提过，*filters/date* 插件可以用来转换你的日志记录中的时间字符串，变成 `LogStash::Timestamp` 对象，然后转存到 `@timestamp` 字段里。

注意：因为在稍后的 **outputs/elasticsearch** 中常用的 **`%{+YYYY.MM.dd}`** 这种写法必须读取 **`@timestamp`** 数据，所以一定不要直接删掉这个字段保留自己的字段，而是应该用 **filters/date** 转换后删除自己的字段！

这在导入旧数据的时候固然非常有用，而在实时数据处理的时候同样有效，因为一般情况下数据流程中我们都会有缓冲区，导致最终的实际处理时间跟事件产生时间略有偏差。

小贴士：个人强烈建议打开 *Nginx* 的 *access_log* 配置项的 *buffer* 参数，对极限响应性能有极大提升！

配置示例

filters/date 插件支持五种时间格式：

ISO8601

类似 "2011-04-19T03:44:01.103Z" 这样的格式。具体Z后面可以有 "08:00"也可以没有，".103"这个也可以没有。常用场景里来说，Nginx 的 *log_format* 配置里就可以使用 `$time_iso8601` 变量来记录请求时间成这种格式。

UNIX

UNIX 时间戳格式，记录的是从 1970 年起始至今的总秒数。Squid 的默认日志格式中就使用了这种格式。

UNIX_MS

这个时间戳则是从 1970 年起始至今的总毫秒数。据我所知，JavaScript 里经常使用这个时间格式。

TAI64N

TAI64N 格式比较少见，是这个样子的： `@4000000052f88ea32489532c` 。我目前只知道常见应用中， qmail 会用这个格式。

Joda-Time 库

Logstash 内部使用了 Java 的 Joda 时间库来作时间处理。所以我们可以使用 Joda 库所支持的时间格式来作具体定义。Joda 时间格式定义见下表：

时间格式

Symbol	Meaning	Presentation	Examples
G	era	text	AD
C	century of era (>=0)	number	20
Y	year of era (>=0)	year	1996
x	weekyear	year	1996
w	week of weekyear	number	27
e	day of week	number	2
E	day of week	text	Tuesday; Tue
y	year	year	1996
D	day of year	number	189
M	month of year	month	July; Jul; 07
d	day of month	number	10
a	halfday of day	text	PM
K	hour of halfday (0~11)	number	0
h	clockhour of halfday (1~12)	number	12
H	hour of day (0~23)	number	0
k	clockhour of day (1~24)	number	24
m	minute of hour	number	30
s	second of minute	number	55
S	fraction of second	number	978
z	time zone	text	Pacific Standard Time; PST
Z	time zone offset/id	zone	-0800; -08:00; America/Los_Angeles
'	escape for text	delimiter	
''	single quote	literal	'

http://joda-time.sourceforge.net/apidocs/org/joda/time/format/DateTimeFormat.html

下面我们写一个 Joda 时间格式的配置作为示例：

```
filter {
    grok {
        match => ["message", "%{HTTPDATE:logdate}"]
    }
    date {
        match => ["logdate", "dd/MMM/yyyy:HH:mm:ss Z"]
    }
}
```

注意：时区偏移量只需要用一个字母 `Z` 即可。

时区问题的解释

很多中国用户经常提一个问题：为什么 @timestamp 比我们晚了 8 个小时？怎么修改成北京时间？

其实，Elasticsearch 内部，对时间类型字段，是统一采用 **UTC** 时间，存成 **long** 长整形数据的！对日志统一采用 UTC 时间存储，是国际安全/运维界的一个通识——欧美公司的服务器普遍广泛分布在多个时区里——不像中国，地域横跨五个时区却只用北京时间。

对于页面查看，ELK 的解决方案是在 Kibana 上，读取浏览器的当前时区，然后在页面上转换时间内容的显示。

所以，建议大家接受这种设定。否则，即便你用 `.getLocalTime` 修改，也还要面临在 Kibana 上反过去修改，以及 Elasticsearch 原有的 `["now-1h" TO "now"]` 这种方便的搜索语句无法正常使用的尴尬。

以上，请读者自行斟酌。

Grok 正则捕获

Grok 是 Logstash 最重要的插件。你可以在 grok 里预定义好命名正则表达式，在稍后(grok参数或者其他正则表达式里)引用它。

正则表达式语法

运维工程师多多少少都会一点正则。你可以在 grok 里写标准的正则，像下面这样：

```
\s+(?<request_time>\d+(?:\.\d+)?)\s+
```

小贴士：这个正则表达式写法对于 *Perl* 或者 *Ruby* 程序员应该很熟悉了，*Python* 程序员可能更习惯写 `(?P<name>pattern)` ，没办法，适应一下吧。

现在给我们的配置文件添加第一个过滤器区段配置。配置要添加在输入和输出区段之间(logstash 执行区段的时候并不依赖于次序，不过为了自己看得方便，还是按次序书写吧)：

```
input {stdin{}}
filter {
    grok {
        match => {
            "message" => "\s+(?<request_time>\d+(?:\.\d+)?)\s+"
        }
    }
}
output {stdout{codec => rubydebug}}
```

运行 logstash 进程然后输入 "begin 123.456 end"，你会看到类似下面这样的输出：

```
{
          "message" => "begin 123.456 end",
         "@version" => "1",
       "@timestamp" => "2014-08-09T11:55:38.186Z",
             "host" => "raochenlindeMacBook-Air.local",
     "request_time" => "123.456"
}
```

漂亮！不过数据类型好像不太满意……*request_time* 应该是数值而不是字符串。

我们已经提过稍后会学习用 `LogStash::Filters::Mutate` 来转换字段值类型，不过在 grok 里，其实有自己的魔法来实现这个功能！

Grok 表达式语法

Grok 支持把预定义的 *grok* 表达式 写入到文件中，官方提供的预定义 grok 表达式见：https://github.com/logstash-plugins/logstash-patterns-core/tree/master/patterns。

注意：在新版本的**logstash**里面，**pattern**目录已经为空，最后一个**commit**提示**core patterns**将会由**logstash-patterns-core gem**来提供，该目录可供用户存放自定义**patterns**

下面是从官方文件中摘抄的最简单但是足够说明用法的示例：

```
USERNAME [a-zA-Z0-9._-]+
USER %{USERNAME}
```

第一行，用普通的正则表达式来定义一个 **grok** 表达式；第二行，通过打印赋值格式**(sprintf format)**，用前面定义好的 **grok** 表达式来定义另一个 **grok** 表达式。

grok 表达式的打印赋值格式的完整语法是下面这样的：

```
%{PATTERN_NAME:capture_name:data_type}
```

小贴士：*data_type* 目前只支持两个值：`int` 和 `float`。

所以我们可以改进我们的配置成下面这样：

```
filter {
    grok {
        match => {
            "message" => "%{WORD} %{NUMBER:request_time:float} %
{WORD}"
        }
    }
}
```

重新运行进程然后可以得到如下结果：

```
{
         "message" => "begin 123.456 end",
        "@version" => "1",
      "@timestamp" => "2014-08-09T12:23:36.634Z",
            "host" => "raochenlindeMacBook-Air.local",
    "request_time" => 123.456
}
```

这次 *request_time* 变成数值类型了。

最佳实践

实际运用中，我们需要处理各种各样的日志文件，如果你都是在配置文件里各自写一行自己的表达式，就完全不可管理了。所以，我们建议是把所有的 grok 表达式统一写入到一个地方。然后用 *filter/grok* 的 `patterns_dir` 选项来指明。

如果你把 "message" 里所有的信息都 grok 到不同的字段了，数据实质上就相当于是重复存储了两份。所以你可以用 `remove_field` 参数来删除掉 *message* 字段，或者用 `overwrite` 参数来重写默认的 *message* 字段，只保留最重要的部分。

重写参数的示例如下：

```
filter {
    grok {
        patterns_dir => ["/path/to/your/own/patterns"]
        match => {
            "message" => "%{SYSLOGBASE} %{DATA:message}"
        }
        overwrite => ["message"]
    }
}
```

更多有关 grok 正则性能的最佳实践(`timeout_millis` 等)，见：https://www.elastic.co/blog/do-you-grok-grok

小贴士

多行匹配

在和 *codec/multiline* 搭配使用的时候，需要注意一个问题，grok 正则和普通正则一样，默认是不支持匹配回车换行的。就像你需要 `=~ //m` 一样也需要单独指定，具体写法是在表达式开始位置加 `(?m)` 标记。如下所示：

```
match => {
    "message" => "(?m)\s+(?<request_time>\d+(?:\.\d+)?)\s+"
}
```

多项选择

有时候我们会碰上一个日志有多种可能格式的情况。这时候要写成单一正则就比较困难，或者全用 `|` 隔开又比较丑陋。这时候，logstash 的语法提供给我们一个有趣的解决方式。

文档中，都说明 logstash/filters/grok 插件的 `match` 参数应该接受的是一个 Hash 值。但是因为早期的 logstash 语法中 Hash 值也是用 `[]` 这种方式书写的，所以其实现在传递 Array 值给 `match` 参数也完全没问题。所以，我们这里其实可以传递多个正则来匹配同一个字段：

```
match => [
    "message", "(?<request_time>\d+(?:\.\d+)?)",
    "message", "%{SYSLOGBASE} %{DATA:message}",
    "message", "(?m)%{WORD}"
]
```

logstash 会按照这个定义次序依次尝试匹配，到匹配成功为止。虽说效果跟用 `|` 分割写个大大的正则是一样的，但是可阅读性好了很多。

最后也是最关键的，我强烈建议每个人都要使用 Grok Debugger 来调试自己的 **grok** 表达式。



dissect

grok 作为 Logstash 最广为人知的插件，在性能和资源损耗方面同样也广为诟病。为了应对这个情况，同时也考虑到大多数时候，日志格式并没有那么复杂，Logstash 开发团队在 5.0 版新添加了另一个解析字段的插件：dissect。

当日志格式有比较简明的分隔标志位，而且重复性较大的时候，我们可以使用 dissect 插件更快的完成解析工作。下面是解析 syslog 的示例：

示例

```
filter {
    dissect {
        mapping => {
            "message" => "%{ts} %{+ts} %{+ts} %{src} %{} %{prog}
[%{pid}]: %{msg}"
        }
        convert_datatype => {
            pid => "int"
        }
    }
}
```

语法解释

我们看到上面使用了和 Grok 很类似的 `%{}` 语法来表示字段，这显然是基于习惯延续的考虑。不过示例中 `%{+ts}` 的加号就不一般了。dissect 除了字段外面的字符串定位功能以外，还通过几个特殊符号来处理字段提取的规则：

- %{+key} 这个 `+` 表示，前面已经捕获到一个 key 字段了，而这次捕获的内容，自动添补到之前 key 字段内容的后面。
- %{+key/2} 这个 `/2` 表示，在有多次捕获内容都填到 key 字段里的时候，拼接字符串的顺序谁前谁后。`/2` 表示排第 2 位。
- %{?string} 这个 `?` 表示，这块只是一个占位，并不会实际生成捕获字段存到 Event 里面。

- %{?string} %{&string} 当同样捕获名称都是 string，但是一个 `?` 一个 `&` 的时候，表示这是一个键值对。

比如对 `http://rizhiyi.com/index.do?id=123` 写这么一段配置：

```
http://%{domain}/%{?url}?%{?arg1}=%{&arg1}
```

则最终生成的 Event 内容是这样的：

```
{
  domain => "rizhiyi.com",
  id => "123"
}
```

GeoIP 地址查询归类

GeoIP 是最常见的免费 IP 地址归类查询库，同时也有收费版可以采购。GeoIP 库可以根据 IP 地址提供对应的地域信息，包括国别，省市，经纬度等，对于可视化地图和区域统计非常有用。

配置示例

```
filter {
    geoip {
        source => "message"
    }
}
```

运行结果

```
{
        "message" => "183.60.92.253",
       "@version" => "1",
     "@timestamp" => "2014-08-07T10:32:55.610Z",
           "host" => "raochenlindeMacBook-Air.local",
          "geoip" => {
                      "ip" => "183.60.92.253",
           "country_code2" => "CN",
           "country_code3" => "CHN",
            "country_name" => "China",
          "continent_code" => "AS",
             "region_name" => "30",
               "city_name" => "Guangzhou",
                "latitude" => 23.11670000000001,
               "longitude" => 113.25,
                "timezone" => "Asia/Chongqing",
        "real_region_name" => "Guangdong",
                "location" => [
            [0] 113.25,
            [1] 23.11670000000001
        ]
    }
}
```

配置说明

GeoIP 库数据较多，如果你不需要这么多内容，可以通过 `fields` 选项指定自己所需要的。下例为全部可选内容：

```
filter {
    geoip {
        fields => ["city_name", "continent_code", "country_code2
", "country_code3", "country_name", "dma_code", "ip", "latitude"
, "longitude", "postal_code", "region_name", "timezone"]
    }
}
```

需要注意的是：`geoip.location` 是 logstash 通过 `latitude` 和 `longitude` 额外生成的数据。所以，如果你是想要经纬度又不想重复数据的话，应该像下面这样做：

filter { geoip { fields => ["city_name", "country_code2", "country_name", "latitude", "longitude", "region_name"] remove_field => ["[geoip][latitude]", "[geoip][longitude]"] } } ```

小贴士

geoip 插件的 "source" 字段可以是任一处理后的字段，比如 "client_ip"，但是字段内容却需要小心！geoip 库内只存有公共网络上的 IP 信息，查询不到结果的，会直接返回 null，而 logstash 的 geoip 插件对 null 结果的处理是：不生成对应的 **geoip.**字段。

所以读者在测试时，如果使用了诸如 127.0.0.1, 172.16.0.1, 192.168.0.1, 10.0.0.1 等内网地址，会发现没有对应输出！

JSON 编解码

在上一章，已经讲过在 codec 中使用 JSON 编码。但是，有些日志可能是一种复合的数据结构，其中只是一部分记录是 JSON 格式的。这时候，我们依然需要在 filter 阶段，单独启用 JSON 解码插件。

配置示例

```
filter {
    json {
        source => "message"
        target => "jsoncontent"
    }
}
```

运行结果

```
{
    "@version": "1",
    "@timestamp": "2014-11-18T08:11:33.000Z",
    "host": "web121.mweibo.tc.sinanode.com",
    "message": "{\"uid\":3081609001,\"type\":\"signal\"}",
    "jsoncontent": {
        "uid": 3081609001,
        "type": "signal"
    }
}
```

小贴士

如果不打算使用多层结构的话，删掉 `target` 配置即可。新的结果如下：

```
{
    "@version": "1",
    "@timestamp": "2014-11-18T08:11:33.000Z",
    "host": "web121.mweibo.tc.sinanode.com",
    "message": "{\"uid\":3081609001,\"type\":\"signal\"}",
    "uid": 3081609001,
    "type": "signal"
}
```

Key-Value 切分

在很多情况下，日志内容本身都是一个类似于 key-value 的格式，但是格式具体的样式却是多种多样的。logstash 提供 `filters/kv` 插件，帮助处理不同样式的 key-value 日志，变成实际的 LogStash::Event 数据。

配置示例

```
filter {
    ruby {
        init => "@kname = ['method','uri','verb']"
        code => "
            new_event = LogStash::Event.new(Hash[@kname.zip(even
t.get('request').split('|'))])
            new_event.remove('@timestamp')
            event.append(new_event)
        "
    }
    if [uri] {
        ruby {
            init => "@kname = ['url_path','url_args']"
            code => "
                new_event = LogStash::Event.new(Hash[@kname.zip(
event.get('uri').split('?'))])
                new_event.remove('@timestamp')
                event.append(new_event)
            "
        }
        kv {
            prefix => "url_"
            source => "url_args"
            field_split => "&"
            remove_field => [ "url_args", "uri", "request" ]
        }
    }
}
```

解释

Nginx 访问日志中的 `$request`，通过这段配置，可以详细切分成 `method`，`url_path`，`verb`，`url_a`，`url_b` ...

进一步的，如果 url_args 中有过多字段，可能导致 Elasticsearch 集群因为频繁 update mapping 或者消耗太多内存在 cluster state 上而宕机。所以，更优的选择，是只保留明确有用的 url_args 内容，其他部分舍去。

```
kv {
    prefix => "url_"
    source => "url_args"
    field_split => "&"
    include_keys => [ "uid", "cip" ]
    remove_field => [ "url_args", "uri", "request" ]
}
```

上例即表示，除了 `url_uid` 和 `url_cip` 两个字段以外，其他的 `url_*` 都不保留。

数值统计(Metrics)

filters/metrics 插件是使用 Ruby 的 *Metriks* 模块来实现在内存里实时的计数和采样分析。该模块支持两个类型的数值分析：meter 和 timer。下面分别举例说明：

Meter 示例(速率阈值检测)

web 访问日志的异常状态码频率是运维人员会非常关心的一个数据。通常我们的做法，是通过 logstash 或者其他日志分析脚本，把计数发送到 rrdtool 或者 graphite 里面。然后再通过 check-graphite 脚本之类的东西来检查异常并报警。

事实上这个事情可以直接在 logstash 内部就完成。比如如果最近一分钟 504 请求的个数超过 100 个就报警：

```
filter {
    metrics {
        meter => "error_%{status}"
        add_tag => "metric"
        ignore_older_than => 10
    }
    if "metric" in [tags] {
        ruby {
            code => "event.cancel if (event.get('[error_504][rat
e_1m]') * 60 > 100)"
        }
    }
}
output {
    if "metric" in [tags] {
        exec {
            command => "echo \"Out of threshold: %{[error_504][r
ate_1m]}\""
        }
    }
}
```

这里需要注意 `*60` 的含义。

metriks 模块生成的 *rate_1m/5m/15m* 意思是：最近 1，5，15 分钟的每秒速率！

Timer 示例(box and whisker 异常检测)

官版的 *filters/metrics* 插件只适用于 metric 事件的检查。由插件生成的新事件内部不存有来自 input 区段的实际数据信息。所以，要完成我们的百分比分布箱体检测，需要首先对代码稍微做几行变动，即在 metric 的 timer 事件里加一个属性，存储最近一个实际事件的数值：https://github.com/chenryn/logstash/commit/bc7bf34caf551d8a149605cf28e7c5d33fae7458

有了这个 last 值，然后我们就可以用如下配置来探测异常数据了：

```
filter {
    metrics {
        timer => {"rt" => "%{request_time}"}
        percentiles => [25, 75]
        add_tag => "percentile"
    }
    if "percentile" in [tags] {
        ruby {
            code => "l=event.get('[rt][p75]')-event.get('[rt][p25]');event.set('[rt][low]', event.get('[rt][p25]')-l);event.set('[rt][high]',event.get('[rt][p75]')+l)"
        }
    }
}
output {
    if "percentile" in [tags] and ([rt][last] > [rt][high] or [rt][last] < [rt][low]) {
        exec {
            command => "echo \"Anomaly: %{[rt][last]}\""
        }
    }
}
```

小贴士：有关 *box and shisker plot* 内容和重要性，参见《数据之魅》一书。

数据修改(Mutate)

filters/mutate 插件是 Logstash 另一个重要插件。它提供了丰富的基础类型数据处理能力。包括类型转换，字符串处理和字段处理等。

类型转换

类型转换是 *filters/mutate* 插件最初诞生时的唯一功能。其应用场景在之前 Codec/JSON 小节已经提到。

可以设置的转换类型包括："integer"，"float" 和 "string"。示例如下：

```
filter {
    mutate {
        convert => ["request_time", "float"]
    }
}
```

注意：**mutate** 除了转换简单的字符值，还支持对数组类型的字段进行转换，即将 `["1","2"]` 转换成 `[1,2]`。但不支持对哈希类型的字段做类似处理。有这方面需求的可以采用稍后讲述的 **filters/ruby** 插件完成。

字符串处理

- gsub

仅对字符串类型字段有效

```
    gsub => ["urlparams", "[\\?#]", "_"]
```

- split

```
filter {
    mutate {
        split => ["message", "|"]
    }
}
```

随意输入一串以 `|` 分割的字符，比如 "123|321|adfd|dfjld*=123"，可以看到如下输出：

```
{
    "message" => [
        [0] "123",
        [1] "321",
        [2] "adfd",
        [3] "dfjld*=123"
    ],
    "@version" => "1",
    "@timestamp" => "2014-08-20T15:58:23.120Z",
    "host" => "raochenlindeMacBook-Air.local"
}
```

- join

仅对数组类型字段有效

我们在之前已经用 `split` 割切的基础再 `join` 回去。配置改成：

```
filter {
    mutate {
        split => ["message", "|"]
    }
    mutate {
        join => ["message", ","]
    }
}
```

filter 区段之内，是顺序执行的。所以我们最后看到的输出结果是：

```
{
    "message" => "123,321,adfd,dfjld*=123",
    "@version" => "1",
    "@timestamp" => "2014-08-20T16:01:33.972Z",
    "host" => "raochenlindeMacBook-Air.local"
}
```

- merge

合并两个数组或者哈希字段。依然在之前 split 的基础上继续：

```
filter {
    mutate {
        split => ["message", "|"]
    }
    mutate {
        merge => ["message", "message"]
    }
}
```

我们会看到输出：

```
{
       "message" => [
        [0] "123",
        [1] "321",
        [2] "adfd",
        [3] "dfjld*=123",
        [4] "123",
        [5] "321",
        [6] "adfd",
        [7] "dfjld*=123"
    ],
      "@version" => "1",
    "@timestamp" => "2014-08-20T16:05:53.711Z",
          "host" => "raochenlindeMacBook-Air.local"
}
```

如果 src 字段是字符串，会自动先转换成一个单元素的数组再合并。把上一示例中的来源字段改成 "host"：

```
filter {
    mutate {
        split => ["message", "|"]
    }
    mutate {
        merge => ["message", "host"]
    }
}
```

结果变成：

```
{
       "message" => [
        [0] "123",
        [1] "321",
        [2] "adfd",
        [3] "dfjld*=123",
        [4] "raochenlindeMacBook-Air.local"
    ],
      "@version" => "1",
    "@timestamp" => "2014-08-20T16:07:53.533Z",
          "host" => [
        [0] "raochenlindeMacBook-Air.local"
    ]
}
```

看，目的字段 "message" 确实多了一个元素，但是来源字段 "host" 本身也由字符串类型变成数组类型了！

下面你猜，如果来源位置写的不是字段名而是直接一个字符串，会产生什么奇特的效果呢？

- strip
- lowercase
- uppercase

字段处理

- rename

重命名某个字段，如果目的字段已经存在，会被覆盖掉：

```
filter {
    mutate {
        rename => ["syslog_host", "host"]
    }
}
```

- update

更新某个字段的内容。如果字段不存在，不会新建。

- replace

作用和 update 类似，但是当字段不存在的时候，它会起到 `add_field` 参数一样的效果，自动添加新的字段。

执行次序

需要注意的是，filter/mutate 内部是有执行次序的。其次序如下：

```
    rename(event) if @rename
    update(event) if @update
    replace(event) if @replace
    convert(event) if @convert
    gsub(event) if @gsub
    uppercase(event) if @uppercase
    lowercase(event) if @lowercase
    strip(event) if @strip
    remove(event) if @remove
    split(event) if @split
    join(event) if @join
    merge(event) if @merge

    filter_matched(event)
```

而 `filter_matched` 这个 filters/base.rb 里继承的方法也是有次序的。

```
  @add_field.each do |field, value|
  end
  @remove_field.each do |field|
  end
  @add_tag.each do |tag|
  end
  @remove_tag.each do |tag|
  end
```

随心所欲的 Ruby 处理

如果你稍微懂那么一点点 Ruby 语法的话，*filters/ruby* 插件将会是一个非常有用的工具。

比如你需要稍微修改一下 `LogStash::Event` 对象，但是又不打算为此写一个完整的插件，用 *filters/ruby* 插件绝对感觉良好。

配置示例

```
filter {
    ruby {
        init => "@kname = ['client','servername','url','status',
'time','size','upstream','upstreamstatus','upstreamtime','refere
r','xff','useragent']"
        code => "
            new_event = LogStash::Event.new(Hash[@kname.zip(even
t.get('message').split('|'))])
            new_event.remove('@timestamp')
            event.append(new_event)"
    }
}
```

官网示例是一个比较有趣但是没啥大用的做法 —— 随机取消 90% 的事件。

所以上面我们给出了一个有用而且强大的实例。

解释

通常我们都是用 *filters/grok* 插件来捕获字段的，但是正则耗费大量的 CPU 资源，很容易成为 Logstash 进程的瓶颈。

而实际上，很多流经 Logstash 的数据都是有自己预定义的特殊分隔符的，我们可以很简单的直接切割成多个字段。

filters/mutate 插件里的 "split" 选项只能切成数组，后续很不方便使用和识别。而在 *filters/ruby* 里，我们可以通过 "init" 参数预定义好由每个新字段的名字组成的数组，然后在 "code" 参数指定的 Ruby 语句里通过两个数组的 zip 操作生成一个哈希并添加进数组里。短短一行 Ruby 代码，可以减少 50% 以上的 CPU 使用率。

注1：从 Logstash-2.3 开始， `LogStash::Event.append` 不再直接接受 Hash 对象，而必须是 `LogStash::Event` 对象。所以示例变成要先初始化一个新 event，再把无用的 `@timestamp` 移除，再 append 进去。否则会把 `@timestamp` 变成有两个时间的数组了！

注2：从 Logstash-5.0 开始， `LogStash::Event` 改为 Java 实现，直接使用 `event["parent"]["child"]` 形式获取的不是原事件的引用而是复制品。需要改用 `event.get('[parent][child]')` 和 `event.set('[parent][child]', 'value')` 的方法。

filters/ruby 插件用途远不止这一点，下一节你还会继续见到它的身影。

更多实例

2014 年 09 年 23 日新增

```
filter{
    date {
        match => ["datetime" , "UNIX"]
    }
    ruby {
        code => "event.cancel if 5 * 24 * 3600 < (event['@timest
amp']-::Time.now).abs"
    }
}
```

在实际运用中，我们几乎肯定会碰到出乎意料的输入数据。这都有可能导致 Elasticsearch 集群出现问题。

当数据格式发生变化，比如 UNIX 时间格式变成 UNIX_MS 时间格式，会导致 logstash 疯狂创建新索引，集群崩溃。

或者误输入过老的数据时，因为一般我们会 close 几天之前的索引以节省内存，必要时再打开。而直接尝试把数据写入被关闭的索引会导致内存问题。

这时候我们就需要提前校验数据的合法性。上面配置，就是用于过滤掉时间范围与当前时间差距太大的非法数据的。

split 拆分事件

上一章我们通过 multiline 插件将多行数据合并进一个事件里，那么反过来，也可以把一行数据，拆分成多个事件。这就是 split 插件。

配置示例

```
filter {
    split {
        field => "message"
        terminator => "#"
    }
}
```

运行结果

这个测试中，我们在 intputs/stdin 的终端中输入一行数据："test1#test2"，结果看到输出两个事件：

```
{
    "@version": "1",
    "@timestamp": "2014-11-18T08:11:33.000Z",
    "host": "web121.mweibo.tc.sinanode.com",
    "message": "test1"
}
{
    "@version": "1",
    "@timestamp": "2014-11-18T08:11:33.000Z",
    "host": "web121.mweibo.tc.sinanode.com",
    "message": "test2"
}
```

重要提示

split 插件中使用的是 yield 功能，其结果是 split 出来的新事件，会直接结束其在 filter 阶段的历程，也就是说写在 split 后面的其他 filter 插件都不起作用，进入到 output 阶段。所以，一定要保证 **split** 配置写在全部 **filter** 配置的最后。

使用了类似功能的还有 clone 插件。

注：从 *logstash-1.5.0beta1* 版本以后修复该问题。

elapsed

```
filter {
  grok {
    match => ["message", "%{TIMESTAMP_ISO8601} START id: (?<task
_id>.*)"]
    add_tag => [ "taskStarted" ]
  }
  grok {
    match => ["message", "%{TIMESTAMP_ISO8601} END id: (?<task_i
d>.*)"]
    add_tag => [ "taskTerminated"]
  }
  elapsed {
    start_tag => "taskStarted"
    end_tag => "taskTerminated"
    unique_id_field => "task_id"
  }
}
```

保存进 Elasticsearch

Logstash 可以使用不同的协议实现完成将数据写入 Elasticsearch 的工作。在不同时期，也有不同的插件实现方式。本节以最新版为准，即主要介绍 HTTP 方式。同时也附带一些原有的 node 和 transport 方式的介绍。

配置示例

```
output {
    elasticsearch {
        hosts => ["192.168.0.2:9200"]
        index => "logstash-%{type}-%{+YYYY.MM.dd}"
        document_type => "%{type}"
        flush_size => 20000
        idle_flush_time => 10
        sniffing => true
        template_overwrite => true
    }
}
```

解释

批量发送

在过去的版本中，主要由本插件的 `flush_size` 和 `idle_flush_time` 两个参数共同控制 Logstash 向 Elasticsearch 发送批量数据的行为。以上面示例来说：Logstash 会努力攒到 20000 条数据一次性发送出去，但是如果 10 秒钟内也没攒够 20000 条，Logstash 还是会以当前攒到的数据量发一次。

默认情况下，`flush_size` 是 500 条，`idle_flush_time` 是 1 秒。这也是很多人改大了 `flush_size` 也没能提高写入 ES 性能的原因——Logstash 还是 1 秒钟发送一次。

从 5.0 开始，这个行为有了另一个前提：`flush_size` 的大小不能超过 Logstash 运行时的命令行参数设置的 `batch_size`，否则将以 `batch_size` 为批量发送的大小。

索引名

写入的 ES 索引的名称，这里可以使用变量。为了更贴合日志场景，Logstash 提供了 `%{+YYYY.MM.dd}` 这种写法。在语法解析的时候，看到以 + 号开头的，就会自动认为后面是时间格式，尝试用时间格式来解析后续字符串。所以，之前处理过程中不要给自定义字段取个加号开头的名字……

此外，注意索引名中不能有大写字母，否则 ES 在日志中会报 *InvalidIndexNameException*，但是 Logstash 不会报错，这个错误比较隐晦，也容易掉进这个坑中。

轮询

Logstash 1.4.2 在 transport 和 http 协议的情况下是固定连接指定 host 发送数据。从 1.5.0 开始，host 可以设置数组，它会从节点列表中选取不同的节点发送数据，达到 Round-Robin 负载均衡的效果。

不同版本的协议沿革

1.4.0 版本之前，有 `logstash-output-elasticsearch`, `logstash-output-elasticsearch_http`, `logstash-output-elasticsearch_river` 三个插件。

1.4.0 到 2.0 版本之间，配合 Elasticsearch 废弃 river 方法，只剩下 `logstash-output-elasticsearch` 一个插件，同时实现了 node、transport、http 三种协议。

2.0 版本开始，为了兼容性和调试方便，`logstash-output-elasticsearch` 改为只支持 *http* 协议。想继续使用 *node* 或者 *transport* 协议的用户，需要单独安装 `logstash-output-elasticsearch_java` 插件。

一个小集群里，使用 *node* 协议最方便了。Logstash 以 elasticsearch 的 client 节点身份(即不存数据不参加选举)运行。如果你运行下面这行命令，你就可以看到自己的 logstash 进程名，对应的 `node.role` 值是 **c**：

```
# curl 127.0.0.1:9200/_cat/nodes?v
host       ip          heap.percent ram.percent load node.role maste
r name
local 192.168.0.102   7         c           -       logstash-local-1036
-2012
local 192.168.0.2     7         d           *       Sunstreak
```

Logstash-1.5 以后，也不再分发一个内嵌的 elasticsearch 服务器。如果你想变更 node 协议下的这些配置，在 `$PWD/elasticsearch.yml` 文件里写自定义配置即可，logstash 会尝试自动加载这个文件。

对于拥有很多索引的大集群，你可以用 *transport* 协议。logstash 进程会转发所有数据到你指定的某台主机上。这种协议跟上面的 *node* 协议是不同的。*node* 协议下的进程是可以接收到整个 Elasticsearch 集群状态信息的，当进程收到一个事件时，它就知道这个事件应该存在集群内哪个机器的分片里，所以它就会直接连接该机器发送这条数据。而 *transport* 协议下的进程不会保存这个信息，在集群状态更新(节点变化，索引变化都会发送全量更新)时，就不会对所有的 logstash 进程也发送这种信息。更多 Elasticsearch 集群状态的细节，参阅本书后续章节。

小贴士

- 经常有同学问，为什么 Logstash 在有多个 conf 文件的情况下，进入 ES 的数据会重复，几个 conf 数据就会重复几次。其实问题原因在之前启动参数章节有提过，output 段顺序执行，没有对日志 type 进行判断的各插件配置都会全部执行一次。在 output 段对 type 进行判断的语法如下所示：

  ```
  output {
  if [type] == "nginxaccess" {
    elasticsearch { }
  }
  }
  ```

模板

Elasticsearch 支持给索引预定义设置和 mapping(前提是你用的 elasticsearch 版本支持这个 API，不过估计应该都支持)。Logstash 自带有一个优化好的模板，内容如下:

```
{
    "template" : "logstash-*",
    "version" : 50001,
    "settings" : {
        "index.refresh_interval" : "5s"
    },
    "mappings" : {
        "_default_" : {
            "_all" : {"enabled" : true, "norms" : false},
            "dynamic_templates" : [ {
                "message_field" : {
                    "path_match" : "message",
                    "match_mapping_type" : "string",
                    "mapping" : {
                        "type" : "text",
                        "norms" : false
                    }
                }
            }, {
                "string_fields" : {
                    "match" : "*",
                    "match_mapping_type" : "string",
                    "mapping" : {
                        "type" : "text", "norms" : false,
                        "fields" : {
                            "keyword" : { "type": "keyword"  }
                        }
                    }
                }
            }  ],
            "properties" : {
                "@timestamp": { "type": "date", "include_in_all"
: false  },
                "@version": { "type": "keyword", "include_in_all"
: false  },
                "geoip"  : {
                    "dynamic": true,
                    "properties" : {
                        "ip": { "type": "ip"  },
```

```
                    "location" : { "type" : "geo_point"  },
                    "latitude" : { "type" : "half_float"  },
                    "longitude" : { "type" : "half_float"  }
                }
            }
        }
    }
  }
}
```

这其中的关键设置包括：

- template for index-pattern

只有匹配 `logstash-*` 的索引才会应用这个模板。有时候我们会变更 Logstash 的默认索引名称，记住你也得通过 PUT 方法上传可以匹配你自定义索引名的模板。当然，我更建议的做法是，把你自定义的名字放在 "logstash-" 后面，变成 `index => "logstash-custom-%{+yyyy.MM.dd}"` 这样。

- refresh_interval for indexing

Elasticsearch 是一个近实时搜索引擎。它实际上是每 1 秒钟刷新一次数据。对于日志分析应用，我们用不着这么实时，所以 logstash 自带的模板修改成了 5 秒钟。你还可以根据需要继续放大这个刷新间隔以提高数据写入性能。

- multi-field with keyword

Elasticsearch 会自动使用自己的默认分词器(空格，点，斜线等分割)来分析字段。分词器对于搜索和评分是非常重要的，但是大大降低了索引写入和聚合请求的性能。所以 logstash 模板定义了一种叫"多字段"(multi-field)类型的字段。这种类型会自动添加一个 ".keyword" 结尾的字段，并给这个字段设置为不启用分词器。简单说，你想获取 url 字段的聚合结果的时候，不要直接用 "url" ，而是用 "url.keyword" 作为字段名。当你还对分词字段发起聚合和排序请求的时候，直接提示无法构建 fielddata 了！

在 Logstash 5.0 中，同时还保留携带了针对 Elasticsearch 2.x 的 template 文件，在那里，通过旧版本的 mapping 配置，达到和新版本相同的行为效果：对应统计字段明确设置 `"index":"not_analyzed","doc_values":true` ，以及对分词字段加上对 fielddata 的 `{"format":"disabled"}` 。

- geo_point

Elasticsearch 支持 *geo_point* 类型， *geo distance* 聚合等等。比如说，你可以请求某个 *geo_point* 点方圆 10 千米内数据点的总数。在 Kibana 的 tilemap 类型面板里，就会用到这个类型的数据。

- half_float

Elasticsearch 5.0 新引入了 *half_float* 类型。比标准的 *float* 类型占用更少的资源，提供更好的性能。在明确自己数值范围较小的时候可用。刚巧，经纬度就是一个明确的数值范围很小的数据。

其他模板配置建议

- order

如果你有自己单独定制 template 的想法，很好。这时候有几种选择：

1. 在 logstash/outputs/elasticsearch 配置中开启 `manage_template => false` 选项，然后一切自己动手；
2. 在 logstash/outputs/elasticsearch 配置中开启 `template => "/path/to/your/tmpl.json"` 选项，让 logstash 来发送你自己写的 template 文件；
3. 避免变更 logstash 里的配置，而是另外发送一个 template ，利用 elasticsearch 的 templates order 功能。

这个 order 功能，就是 elasticsearch 在创建一个索引的时候，如果发现这个索引同时匹配上了多个 template ，那么就会先应用 order 数值小的 template 设置，然后再应用一遍 order 数值高的作为覆盖，最终达到一个 merge 的效果。

比如，对上面这个模板已经很满意，只想修改一下 `refresh_interval` ，那么只需要新写一个：

```
{
  "order" : 1,
  "template" : "logstash-*",
  "settings" : {
    "index.refresh_interval" : "20s"
  }
}
```

然后运行 `curl -XPUT http://localhost:9200/_template/template_newid -d '@/path/to/your/tmpl.json'` 即可。

logstash 默认的模板， order 是 0，id 是 logstash，通过 logstash/outputs/elasticsearch 的配置选项 `template_name` 修改。你的新模板就不要跟这个名字冲突了。

发送邮件(Email)

配置示例

```
output {
    email {
        to => "admin@website.com,root@website.com"
        cc => "other@website.com"
        via => "smtp"
        subject => "Warning: %{title}"
        options => {
            smtpIporHost       => "localhost",
            port               => 25,
            domain             => 'localhost.localdomain',
            userName           => nil,
            password           => nil,
            authenticationType => nil, # (plain, login and cram_
md5)
            starttls           => true
        }
        htmlbody => ""
        body => ""
        attachments => ["/path/to/filename"]
    }
}
```

注意：以上示例适用于 `Logstash 1.5` ，`options =>` 的参数配置在 `Logstash 2.0` 之后的版本已被移除，（*126* 邮箱发送到 *qq* 邮箱）示例如下：

```
output {
    email {
        port           =>    "25"
        address        =>    "smtp.126.com"
        username       =>    "test@126.com"
        password       =>    ""
        authentication =>    "plain"
        use_tls        =>    true
        from           =>    "test@126.com"
        subject        =>    "Warning: %{title}"
        to             =>    "test@qq.com"
        via            =>    "smtp"
        body           =>    "%{message}"
    }
}
```

解释

outputs/email 插件支持 SMTP 协议和 sendmail 两种方式，通过 `via` 参数设置。SMTP 方式有较多的 options 参数可配置。sendmail 只能利用本机上的 sendmail 服务来完成 —— 文档上描述了 Mail 库支持的 sendmail 配置参数，但实际代码中没有相关处理，不要被迷惑了。。。

调用命令执行(Exec)

outputs/exec 插件的运用也非常简单，如下所示，将 logstash 切割成的内容作为参数传递给命令。这样，在每个事件到达该插件的时候，都会触发这个命令的执行。

```
output {
    exec {
        command => "sendsms.pl \"%{message}\" -t %{user}"
    }
}
```

需要注意的是。这种方式是每次都重新开始执行一次命令并退出。本身是比较慢速的处理方式(程序加载，网络建联等都有一定的时间消耗)。最好只用于少量的信息处理场景，比如不适用 nagios 的其他报警方式。示例就是通过短信发送消息。

保存成文件(File)

通过日志收集系统将分散在数百台服务器上的数据集中存储在某中心服务器上，这是运维最原始的需求。早年的 scribed ，甚至直接就把输出的语法命名为 `<store>` 。Logstash 当然也能做到这点。

和 `LogStash::Inputs::File` 不同, `LogStash::Outputs::File` 里可以使用 sprintf format 格式来自动定义输出到带日期命名的路径。

配置示例

```
output {
    file {
        path => "/path/to/%{+yyyy}/%{+MM}/%{+dd}/%{host}.log.gz"
        message_format => "%{message}"
        gzip => true
    }
}
```

解释

使用 *output/file* 插件首先需要注意的就是 `message_format` 参数。插件默认是输出整个 event 的 JSON 形式数据的。这可能跟大多数情况下使用者的期望不符。大家可能只是希望按照日志的原始格式保存就好了。所以需要定义为 `%{message}` ，当然，前提是在之前的 *filter* 插件中，你没有使用 `remove_field` 或者 `update` 等参数删除或修改 `%{message}` 字段的内容。

另一个非常有用的参数是 gzip。gzip 格式是一个非常奇特而友好的格式。其格式包括有：

- 10字节的头，包含幻数、版本号以及时间戳
- 可选的扩展头，如原文件名
- 文件体，包括DEFLATE压缩的数据
- 8字节的尾注，包括CRC-32校验和以及未压缩的原始数据长度

这样 gzip 就可以一段一段的识别出来数据 —— 反过来说，也就是可以一段一段压缩了添加在后面！

这对于我们流式添加数据简直太棒了！

小贴士：你或许见过网络流传的 *parallel* 命令行工具并发处理数据的神奇文档，但在自己用的时候总见不到效果。实际上就是因为：文档中处理的 *gzip* 文件，可以分开处理然后再合并的。

注意

1. 按照 Logstash 标准，其实应该可以把数据格式的定义改在 codec 插件中完成，但是 logstash-output-file 插件内部实现中跳过了 `@codec.decode` 这步，所以 **codec** 设置无法生效！
2. 按照 Logstash 标准，配置参数的值可以使用 event sprintf 格式。但是 logstash-output-file 插件对 `event.sprintf(@path)` 的结果，还附加了一步 `inside_file_root?` 校验(个人猜测是为了防止越权到其他路径)，这个 `file_root` 是通过直接对 path 参数分割 `/` 符号得到的。如果在 sprintf 格式中带有 `/` 符号，那么被切分后的结果就无法正确解析了。

所以，如下所示配置，虽然看起来是正确的，实际效果却不对，正确写法应该是本节之前的配置示例那样。

```
output {
    file {
        path => "/path/to/%{+yyyy/MM/dd}/%{host}.log.gz"
        codec => line {
            format => "%{message}"
        }
    }
}
```

报警到 Nagios

Logstash 中有两个 output 插件是 nagios 有关的。*outputs/nagios* 插件发送数据给本机的 `nagios.cmd` 管道命令文件，*outputs/nagios_nsca* 插件则是 调用 `send_nsca` 命令以 NSCA 协议格式把数据发送给 nagios 服务器(远端或者本地皆可)。

Nagios.Cmd

nagios.cmd 是 nagios 服务器的核心组件。nagios 事件处理和内外交互都是通过这个管道文件来完成的。

使用 CMD 方式，需要自己保证发送的 Logstash 事件符合 nagios 事件的格式。即必须在 *filter* 阶段预先准备好 `nagios_host` 和 `nagios_service` 字段；此外，如果在 *filter* 阶段也准备好 `nagios_annotation` 和 `nagios_level` 字段，这里也会自动转换成 nagios 事件信息。

```
filter {
    if [message] =~ /err/ {
        mutate {
            add_tag => "nagios"
            rename => ["host", "nagios_host"]
            replace => ["nagios_service", "logstash_check_%{type
}"]
        }
    }
}
output {
    if "nagios" in [tags] {
        nagios { }
    }
}
```

如果不打算在 *filter* 阶段提供 `nagios_level` ，那么也可以在该插件中通过参数配置。

所谓 `nagios_level` ，即我们通过 nagios plugin 检查数据时的返回值。其取值范围和含义如下：

- "0"，代表 "OK"，服务正常；
- "1"，代表 "WARNNING"，服务警告，一般 nagios plugin 命令中使用 `-w` 参数设置该阈值；
- "2"，代表 "CRITICAL"，服务危急，一般 nagios plugin 命令中使用 `-c` 参数设置该阈值；
- "3"，代表 "UNKNOWN"，未知状态，一般会在 timeout 等情况下出现。

默认情况下，该插件会以 "CRITICAL" 等级发送报警给 Nagios 服务器。

nagios.cmd 文件的具体位置，可以使用 `command_file` 参数设置。默认位置是 "/var/lib/nagios3/rw/nagios.cmd"。

关于和 nagios.cmd 交互的具体协议说明，有兴趣的读者请阅读 Using external commands in Nagios 一文，这是《Learning Nagios 3.0》书中内容节选。

NSCA

NSCA 是一种标准的 nagios 分布式扩展协议。分布在各机器上的 `send_nsca` 进程主动将监控数据推送给远端 nagios 服务器的 NSCA 进程。

当 Logstash 跟 nagios 服务器没有在同一个主机上运行的时候，就只能通过 NSCA 方式来发送报警了 —— 当然也必须在 Logstash 服务器上安装 `send_nsca` 命令。

nagios 事件所需要的几个属性在上一段中已经有过描述。不过在使用这个插件的时候，不要求提前准备好，而是可以在该插件内部定义参数：

```
output {
    nagios_nsca {
        nagios_host => "%{host}"
        nagios_service => "logstash_check_%{type}"
        nagios_status => "2"
        message_format => "%{@timestamp}: %{message}"
        host => "nagiosserver.domain.com"
    }
}
```

这里请注意， `host` 和 `nagios_host` 两个参数，分别是用来设置 nagios 服务器的地址，和报警信息中有问题的服务器地址。

关于 NSCA 原理，架构和配置说明，还不了解的读者请阅读官方网站 Using NSClient++ from nagios with NSCA 一节。

推荐阅读

除了 nagios 以外，logstash 同样可以发送信息给其他常见监控系统。方式和 nagios 大同小异：

- *outputs/ganglia* 插件通过 UDP 协议，发送 gmetric 型数据给本机/远端的 `gmond` 或者 `gmetad`
- *outputs/zabbix* 插件调用本机的 `zabbix_sender` 命令发送

输出到 Statsd

基础知识

Statsd 最早是 2008 年 Flickr 公司用 Perl 写的针对 Graphite、datadog 等监控数据后端存储开发的前端网络应用，2011 年 Etsy 公司用 node.js 重构。用于接收(默认UDP)、写入、读取和聚合时间序列数据，包括即时值和累积值等。

Graphite 是用 Python 模仿 RRDtools 写的时间序列数据库套件。包括三个部分：

- carbon: 是一个Twisted守护进程，监听处理数据；
- whisper: 存储时间序列的数据库；
- webapp: 一个用 Django 框架实现的网页应用。

Graphite 安装简介

通过如下几步安装 Graphite：

1. 安装 cairo 和 pycairo 库

```
# yum -y install cairo pycairo
```

1. pip 安装

```
# yum install python-devel python-pip
# pip install django django-tagging carbon whisper graphite-web uwsgi
```

1. 配置 Graphite

```
# cd /opt/graphite/webapp/graphite
# cp local_settings.py.example local_settings.py
# python manage.py syncdb
```

修改 `local_settings.py` 中的 `DATABASE` 为设置的 db 信息。

1. 启动 cabon

```
# cd /opt/graphite/conf/
# cp carbon.conf.example carbon.conf
# cp storage-schemas.conf.example storage-schemas.conf
# cd /opt/graphite/
# ./bin/carbon-cache.py start
```

statsd 安装简介

1. Graphite 地址设置

```
# cd /opt/
# git clone git://github.com/etsy/statsd.git
# cd /opt/statsd
# cp exampleConfig.js Config.js
```

根据 Graphite 服务器地址，修改Config.js 中的配置如下：

```
{
  graphitePort: 2003,
  graphiteHost: "10.10.10.124",
  port: 8125,
  backends: [ "./backends/graphite" ]
}
```

1. uwsgi 配置

```
cd /opt/graphite/webapp/graphite
cat > wsgi_graphite.xml <<EOF
<uwsgi>
  <socket>0.0.0.0:8630</socket>
  <workers>2</workers>
  <processes>2</processes>
  <listen>100</listen>
  <chdir>/opt/graphite/webapp/graphite</chdir>
  <pythonpath>..</pythonpath>
  <module>wsgi</module>
  <pidfile>graphite.pid</pidfile>
  <master>true</master>
  <enable-threads>true</enable-threads>
  <logdate>true</logdate>
  <daemonize>/var/log/uwsgi_graphite.log</daemonize>
</uwsgi>
EOF
cp /opt/graphite/conf/graphite.wsgi /opt/graphite/webapp/graphit
e/wsgi.py
```

1. nginx 的 uwsgi 配置

```
cat > /usr/local/nginx/conf/conf.d/graphite.conf <<EOF
server {
  listen 8081;
  server_name graphite;

  access_log /opt/graphite/storage/log/webapp/access.log ;
  error_log /opt/graphite/storage/log/webapp/error.log ;

  location / {
    uwsgi_pass 0.0.0.0:8630;
    include uwsgi_params;
    proxy_connect_timeout 300;
    proxy_send_timeout 300;
    proxy_read_timeout 300;
  }
}
EOF
```

1. 启动

```
# uwsgi -x /opt/graphite/webapp/graphite/wsgi_graphite.xml
# systemctl nginx reload
```

1. 数据测试

```
echo "test.logstash.num:100|c" | nc -w 1 -u $IP $port
```

如果安装配置是正常的，在graphite的左侧metrics->stats->test->logstash->num的表，statsd里面多了numStats等数据。

配置示例

```
output {
    statsd {
        host => "statsdserver.domain.com"
        namespace => "logstash"
        sender => "%{host}"
        increment => ["httpd.response.%{status}"]
    }
}
```

解释

Graphite 以树状结构存储监控数据，所以 statsd 也是如此。所以发送给 statsd 的数据的 key 也一定得是 "first.second.tree.four" 这样的形式。而在 *logstash-output-statsd* 插件中，就会以三个配置参数来拼接成这种形式：

```
    namespace.sender.metric
```

其中 namespace 和 sender 都是直接设置的，而 metric 又分为好几个不同的参数可以分别设置。statsd 支持的 metric 类型如下：

metric 类型

- increment 增量，一个计量周期内，某个数字接收了多少次，比如 nginx 的 status 状态码。

示例语法： `increment => ["nginx.status.%{status}"]`

- decrement

语法同 increment。

- count 对数字的计数，比如，每秒接收一个数字，一个计量周期内，所有数字的和，比如nginx 的 body_bytes_sent。

示例语法： `count => {"nginx.bytes" => "%{bytes}"}`

- gauge

语法同 count。

* set

语法同 count。

* timing 时间范围内，某种数字的最大值，最小值，平均值，比如 nginx 的响应时间 request_time。

语法同 count。

关于这些 metric 类型的详细说明，请阅读 statsd 文档：https://github.com/etsy/statsd/blob/master/docs/metric_types.md。

推荐阅读

* Etsy 发布 nodejs 版本 statsd 的博客：Measure Anything, Measure Everything
* Flickr 发布 statsd 的博客：Counting & Timing

标准输出(Stdout)

和之前 *inputs/stdin* 插件一样，*outputs/stdout* 插件也是最基础和简单的输出插件。同样在这里简单介绍一下，作为输出插件的一个共性了解。

配置示例

```
output {
    stdout {
        codec => rubydebug
        workers => 2
    }
}
```

解释

输出插件统一具有一个参数是 `workers` 。Logstash 为输出做了多线程的准备。

其次是 codec 设置。codec 的作用在之前已经讲过。可能除了 `codecs/multiline` ，其他 codec 插件本身并没有太多的设置项。所以一般省略掉后面的配置区段。换句话说。上面配置示例的完全写法应该是：

```
output {
    stdout {
        codec => rubydebug {
        }
        workers => 2
    }
}
```

单就 *outputs/stdout* 插件来说，其最重要和常见的用途就是调试。所以在不太有效的时候，加上命令行参数 `-vv` 运行，查看更多详细调试信息。

发送网络数据(TCP)

虽然之前我们已经提到过不建议直接使用 LogStash::Inputs::TCP 和 LogStash::Outputs::TCP 做转发工作，不过在实际交流中，发现确实有不少朋友觉得这种简单配置足够使用，因而不愿意多加一层消息队列的。所以，还是把 Logstash 如何直接发送 TCP 数据也稍微提点一下。

配置示例

```
output {
    tcp {
        host  => "192.168.0.2"
        port  => 8888
        codec => json_lines
    }
}
```

配置说明

在收集端采用 tcp 方式发送给远端的 tcp 端口。这里需要注意的是，默认的 codec 选项是 **json**。而远端的 LogStash::Inputs::TCP 的默认 codec 选项却是 **line** ！所以不指定各自的 codec ，对接肯定是失败的。

另外，由于IO BUFFER 的原因，即使是两端共同约定为 **json** 依然无法正常运行，接收端会认为一行数据没结束，一直等待直至自己 OutOfMemory ！

所以，正确的做法是，发送端指定 codec 为 **json_lines** ，这样每条数据后面会加上一个回车，接收端指定 codec 为 **json_lines** 或者 **json** 均可，这样才能正常处理。包括在收集端已经切割好的字段，也可以直接带入收集端使用了。

HDFS

- https://github.com/dstore-dbap/logstash-webhdfs

This plugin based on WebHDFS api of Hadoop, it just POST data to WebHDFS port. So, it's a native Ruby code.

```
output {
    hadoop_webhdfs {
        workers => 2
        server => "your.nameno.de:14000"
        user => "flume"
        path => "/user/flume/logstash/dt=%{+Y}-%{+M}-%{+d}/logst
ash-%{+H}.log"
        flush_size => 500
        compress => "snappy"
        idle_flush_time => 10
        retry_interval => 0.5
    }
}
```

- https://github.com/avishai-ish-shalom/logstash-hdfs

This plugin based on HDFS api of Hadoop, it import java classes like `org.apache.hadoop.fs.FileSystem` etc.

Configuration

```
output {
    hdfs {
        path => "/path/to/output_file.log"
        enable_append => true
    }
}
```

Howto run

```
CLASSPATH=$(find /path/to/hadoop -name '*.jar' | tr '\n' ':'):/e
tc/hadoop/conf:/path/to/logstash-1.1.7-monolithic.jar java logst
ash.runner agent -f conf/hdfs-output.conf -p /path/to/cloned/log
stash-hdfs
```

场景示例

前面虽然介绍了几十个常用的 Logstash 插件的常见配置项。但是过多的选择下，如何组合使用这些插件，依然是一部分用户的幸福难题。本节，列举一些最常见的日志场景，演示一下针对性的组件搭配。希望能给读者带来一点启发。

Nginx 访问日志

grok

Logstash 默认自带了 apache 标准日志的 grok 正则:

```
COMMONAPACHELOG %{IPORHOST:clientip} %{USER:ident} %{NOTSPACE:au
th} \[%{HTTPDATE:timestamp}\] "(?:%{WORD:verb} %{NOTSPACE:reques
t}(?: HTTP/%{NUMBER:httpversion})?|%{DATA:rawrequest})" %{NUMBER
:response} (?:%{NUMBER:bytes}|-)
COMBINEDAPACHELOG %{COMMONAPACHELOG} %{QS:referrer} %{QS:agent}
```

对于 nginx 标准日志格式，可以发现只是最后多了一个 `$http_x_forwarded_for` 变量。所以 nginx 标准日志的 grok 正则定义是：

```
MAINNGINXLOG %{COMBINEDAPACHELOG} %{QS:x_forwarded_for}
```

自定义的日志格式，可以照此修改。

split

nginx 日志因为部分变量中内含空格，所以很多时候只能使用 `%{QS}` 正则来做分割，性能和细度都不太好。如果能自定义一个比较少见的字符作为分隔符，那么处理起来就简单多了。假设定义的日志格式如下：

```
log_format main "$http_x_forwarded_for | $time_local | $request
| $status | $body_bytes_sent | "
                "$request_body | $content_length | $http_referer
 | $http_user_agent | $nuid | "
                "$http_cookie | $remote_addr | $hostname | $upst
ream_addr | $upstream_response_time | $request_time";
```

实际日志如下：

> 117.136.9.248 | 08/Apr/2015:16:00:01 +0800 | POST /notice/newmessage?sign=cba4f614e05db285850cadc696fcdad0&token=JAGQ92Mjs3--gik_b_DsPIQHcyMKYGpD&did=4b749736ac70f12df700b18cd6d051d5&osn=android&osv=4.0.4&appv=3.0.1&net=460-02-2g&longitude=120.393006&latitude=36.178329&ch=360&lp=1&ver=1&ts=1428479998151&im=869736012353958&sw=0&sh=0&la=zh-CN&lm=weixin&dt=vivoS11t HTTP/1.1 | 200 | 132 | abcd-sign-v1://dd03c57f8cb6fcef919ab5df66f2903f:d51asq5yslwnyz5t/{\x22type\x22:4,\x22uid\x22:7567306} | 89 | - | abcd/3.0.1, Android/4.0.4, vivo S11t | nuid=0C0A0A0A01E02455EA7CF47E02FD072C1428480001.157 | - | 10.10.10.13 | bnx02.abcdprivate.com | 10.10.10.22:9999 | 0.022 | 0.022
> 59.50.44.53 | 08/Apr/2015:16:00:01 +0800 | POST /feed/pubList?appv=3.0.3&did=89da72550de488328e2aba5d97850e9f&dt=iPhone6%2C2&im=B48C21F3-487E-4071-9742-DC6D61710888&la=cn&latitude=0.000000&lm=weixin&longitude=0.000000&lp=-1.000000&net=0-0-wifi&osn=iOS&osv=8.1.3&sh=568.000000&sw=320.000000&token=7NobA7asg3Jb6n9o4ETdPXyNNiHwMs4J&ts=1428480001275 HTTP/1.1 | 200 | 983 | abcd-sign-v1://b398870a0b25b29aae65cd553addc43d:72214ee85d7cca22/{\x22nextkey\x22:\x22\x22,\x22uid\x22:\x2213062545\x22,\x22token\x22:\x227NobA7asg3Jb6n9o4ETdPXyNNiHwMs4J\x22} | 139 | - | Shopping/3.0.3 (iPhone; iOS 8.1.3; Scale/2.00) | nuid=0C0A0A0A81DF2455017D548502E48E2E1428480001.154 | nuid=CgoKDFUk34GFVH0BLo7kAg== | 10.10.10.11 | bnx02.abcdprivate.com | 10.10.10.35:9999 | 0.025 | 0.026

然后还可以针对 request 做更细致的切分。比如 URL 参数部分。很明显，URL 参数中的字段，顺序是乱的，第一行，问号之后的第一个字段是 sign，第二行问号之后的第一个字段是 appv，所以需要将字段进行切分，取出每个字段对应的值。官方自带 grok 满足不了要求。最终采用的 logstash 配置如下：

```
filter {
    ruby {
        init => "@kname = ['http_x_forwarded_for','time_local','
request','status','body_bytes_sent','request_body','content_leng
th','http_referer','http_user_agent','nuid','http_cookie','remot
```

```
e_addr','hostname','upstream_addr','upstream_response_time','req
uest_time']"
        code => "
            new_event = LogStash::Event.new(Hash[@kname.zip(even
t.get('message').split('|'))])
            new_event.remove('@timestamp')
            event.append(new_event)
        "
    }
    if [request] {
        ruby {
            init => "@kname = ['method','uri','verb']"
            code => "
                new_event = LogStash::Event.new(Hash[@kname.zip(
event.get('request').split(' '))])
                new_event.remove('@timestamp')
                event.append(new_event)
            "
        }
        if [uri] {
            ruby {
                init => "@kname = ['url_path','url_args']"
                code => "
                    new_event = LogStash::Event.new(Hash[@kname.
zip(event.get('uri').split('?'))])
                    new_event.remove('@timestamp')
                    event.append(new_event)
                "
            }
            kv {
                prefix => "url_"
                source => "url_args"
                field_split => "& "
                remove_field => [ "url_args","uri","request" ]
            }
        }
    }
    mutate {
        convert => [
            "body_bytes_sent" , "integer",
```

```
            "content_length", "integer",
            "upstream_response_time", "float",
            "request_time", "float"
        ]
    }
    date {
        match => [ "time_local", "dd/MMM/yyyy:hh:mm:ss Z" ]
        locale => "en"
    }
}
```

最终结果：

```
{
                   "message" => "1.43.3.188 | 08/Apr/2015:16:00:
01 +0800 | POST /search/suggest?appv=3.0.3&did=dfd5629d705d40079
5f698055806f01d&dt=iPhone7%2C2&im=AC926907-27AA-4A10-9916-C5DC75
F29399&la=cn&latitude=-33.903867&lm=sina&longitude=151.208137&lp
=-1.000000&net=0-0-wifi&osn=iOS&osv=8.1.3&sh=667.000000&sw=375.0
00000&token=_ovaPz6Ue68ybBuhXustPbG-xf1WbsPO&ts=1428480001567 HT
TP/1.1 | 200 | 353 | abcd-sign-v1://a24b478486d3bb92ed89a901541b
60a5:b23e9d2c14fe6755/{\\x22key\\x22:\\x22last\\x22,\\x22offset\
\x22:\\x220\\x22,\\x22token\\x22:\\x22_ovaPz6Ue68ybBuhXustPbG-xf
1WbsPO\\x22,\\x22limit\\x22:\\x2220\\x22} | 148 | - | abcdShoppi
ng/3.0.3 (iPhone; iOS 8.1.3; Scale/2.00) | nuid=0B0A0A0A9A64AF54
F97634640230944E1428480001.113 | nuid=CgoKC1SvZJpkNHb5TpQwAg== |
 10.10.10.11 | bnx02.abcdprivate.com | 10.10.10.26:9999 | 0.070
| 0.071",
                  "@version" => "1",
                "@timestamp" => "2015-04-08T08:00:01.000Z",
                      "type" => "nginxapiaccess",
                      "host" => "blog05.abcdprivate.com",
                      "path" => "/home/nginx/logs/api.access.log
",
      "http_x_forwarded_for" => "1.43.3.188",
                "time_local" => " 08/Apr/2015:16:00:01 +0800",
                    "status" => "200",
           "body_bytes_sent" => 353,
              "request_body" => "abcd-sign-v1://a24b478486d3bb92
```

```
ed89a901541b60a5:b23e9d2c14fe6755/{\\x22key\\x22:\\x22last\\x22,
\\x22offset\\x22:\\x220\\x22,\\x22token\\x22:\\x22_ovaPz6Ue68ybB
uhXustPbG-xf1WbsPO\\x22,\\x22limit\\x22:\\x2220\\x22}",
            "content_length" => 148,
              "http_referer" => "-",
           "http_user_agent" => "abcdShopping/3.0.3 (iPhone; iOS
 8.1.3; Scale/2.00)",
                      "nuid" => "nuid=0B0A0A0A9A64AF54F976346402
30944E1428480001.113",
               "http_cookie" => "nuid=CgoKC1SvZJpkNHb5TpQwAg==",
               "remote_addr" => "10.10.10.11",
                  "hostname" => "bnx02.abcdprivate.com",
             "upstream_addr" => "10.10.10.26:9999",
    "upstream_response_time" => 0.070,
              "request_time" => 0.071,
                    "method" => "POST",
                      "verb" => "HTTP/1.1",
                  "url_path" => "/search/suggest",
                  "url_appv" => "3.0.3",
                   "url_did" => "dfd5629d705d400795f698055806f01
d",
                    "url_dt" => "iPhone7%2C2",
                    "url_im" => "AC926907-27AA-4A10-9916-C5DC75F
29399",
                    "url_la" => "cn",
              "url_latitude" => "-33.903867",
                    "url_lm" => "sina",
             "url_longitude" => "151.208137",
                    "url_lp" => "-1.000000",
                   "url_net" => "0-0-wifi",
                   "url_osn" => "iOS",
                   "url_osv" => "8.1.3",
                    "url_sh" => "667.000000",
                    "url_sw" => "375.000000",
                 "url_token" => "_ovaPz6Ue68ybBuhXustPbG-xf1WbsP
O",
                    "url_ts" => "1428480001567"
}
```

如果 url 参数过多，可以不使用 kv 切割，或者预先定义成 nested object 后，改成数组形式：

```
        if [uri] {
            ruby {
                init => "@kname = ['url_path','url_args']"
                code => "
                    new_event = LogStash::Event.new(Hash[@kname.
zip(event.get('request').split('?'))])
                    new_event.remove('@timestamp')
                    event.append(new_event)
                "
            }
            if [url_args] {
                ruby {
                    init => "@kname = ['key','value']"
                    code => "event.set('nested_args', event.get(
'url_args').split('&').collect {|i| Hash[@kname.zip(i.split('=')
)]})"
                    remove_field => [ "url_args","uri","request"
 ]
                }
            }
        }
```

采用 nested object 的优化原理和 nested object 的使用方式，请阅读稍后 Elasticsearch 调优章节。

json format

自定义分隔符虽好，但是配置写起来毕竟复杂很多。其实对 logstash 来说，nginx 日志还有另一种更简便的处理方式。就是自定义日志格式时，通过手工拼写，直接输出成 JSON 格式：

```
log_format json '{"@timestamp":"$time_iso8601",'
                 '"host":"$server_addr",'
                 '"clientip":"$remote_addr",'
                 '"size":$body_bytes_sent,'
                 '"responsetime":$request_time,'
                 '"upstreamtime":"$upstream_response_time",'
                 '"upstreamhost":"$upstream_addr",'
                 '"http_host":"$host",'
                 '"url":"$uri",'
                 '"xff":"$http_x_forwarded_for",'
                 '"referer":"$http_referer",'
                 '"agent":"$http_user_agent",'
                 '"status":"$status"}';
```

然后采用下面的 logstash 配置即可：

```
input {
    file {
        path => "/var/log/nginx/access.log"
        codec => json
    }
}
filter {
    mutate {
        split => [ "upstreamtime", "," ]
    }
    mutate {
        convert => [ "upstreamtime", "float" ]
    }
}
```

这里采用多个 mutate 插件，因为 upstreamtime 可能有多个数值，所以先切割成数组以后，再分别转换成浮点型数值。而在 mutate 中，convert 函数执行优先级高于 split 函数，所以只能分开两步写。mutate 内各函数优先级顺序，之前插件介绍章节有详细说明，读者可以返回去加强阅读。

syslog

Nginx 从 1.7 版开始，加入了 syslog 支持。Tengine 则更早。这样我们可以通过 syslog 直接发送日志出来。Nginx 上的配置如下：

```
access_log syslog:server=unix:/data0/rsyslog/nginx.sock locallog
;
```

或者直接发送给远程 logstash 机器：

```
access_log syslog:server=192.168.0.2:5140,facility=local6,tag=ng
inx_access,severity=info logstashlog;
```

默认情况下，Nginx 将使用 `local7.info` 等级，`nginx` 为标签，发送数据。注意，采用 syslog 发送日志的时候，无法配置 `buffer=16k` 选项。

Nginx 错误日志

Nginx 错误日志是运维人员最常见但又极其容易忽略的日志类型之一。Nginx 错误日志即没有统一明确的分隔符，也没有特别方便的正则模式，但通过 logstash 不同插件的组合，还是可以轻松做到数据处理。

值得注意的是，Nginx 错误日志中，有一类数据是接收过大请求体时的报错，默认信息会把请求体的具体字节数记录下来。每次请求的字节数基本都是在变化的，这意味着常用的 topN 等聚合函数对该字段都没有明显效果。所以，对此需要做一下特殊处理。

最后形成的 logstash 配置如下所示：

```
filter {
    grok {
        match => { "message" => "(?<datetime>\d\d\d\d/\d\d/\d\d
\d\d:\d\d:\d\d) \[(?<errtype>\w+)\] \S+: \*\d+ (?<errmsg>[^,]+),
 (?<errinfo>.*)$" }
    }
    mutate {
        rename => [ "host", "fromhost" ]
        gsub => [ "errmsg", "too large body: \d+ bytes", "too la
rge body" ]
    }
    if [errinfo]
    {
        ruby {
            code => "
                new_event = LogStash::Event.new(Hash[event.get('
errinfo').split(', ').map{|l| l.split(': ')}])
                new_event.remove('@timestamp')
                event.append(new_event)
            "
        }
    }
    grok {
        match => { "request" => '"%{WORD:verb} %{URIPATH:urlpath
}(?:\?%{NGX_URIPARAM:urlparam})?(?: HTTP/%{NUMBER:httpversion})"
' }
        patterns_dir => ["/etc/logstash/patterns"]
        remove_field => [ "message", "errinfo", "request" ]
    }
}
```

经过这段 logstash 配置的 Nginx 错误日志生成的事件如下所示：

```
{
    "@version": "1",
    "@timestamp": "2015-07-02T01:26:40.000Z",
    "type": "nginx-error",
    "errtype": "error",
    "errmsg": "client intended to send too large body",
    "fromhost": "web033.mweibo.yf.sinanode.com",
    "client": "36.16.7.17",
    "server": "api.v5.weibo.cn",
    "host": "\"api.weibo.cn\"",
    "verb": "POST",
    "urlpath": "/2/client/addlog_batch",
    "urlparam": "gsid=_2A254UNaSDeTxGeRI7FMX9CrEyj2IHXVZRG1arDV6
PUJbrdANLROskWp9bXakjUZM5792FW9A5S9EU4jxqQ..&wm=3333_2001&i=0c6f
156&b=1&from=1053093010&c=iphone&v_p=21&skin=default&v_f=1&s=8f1
4e573&lang=zh_CN&ua=iPhone7,1__weibo__5.3.0__iphone__os8.3",
    "httpversion": "1.1"
}
```

postfix 日志

postfix 是 Linux 平台上最常用的邮件服务器软件。邮件服务的运维复杂度一向较高，在此提供一个针对 postfix 日志的解析处理方案。方案出自：https://github.com/whyscream/postfix-grok-patterns。

因为 postfix 默认通过 syslog 方式输出日志，所以可以选择通过 rsyslog 直接转发给 logstash，也可以由 logstash 读取 rsyslog 记录的文件。下列配置中省略了对 syslog 协议解析的部分。

```
filter {
    # grok log lines by program name (listed alpabetically)
    if [program] =~ /^postfix.*\/anvil$/ {
        grok {
            patterns_dir   => ["/etc/logstash/patterns.d"]
            match          => [ "message", "%{POSTFIX_ANVIL}" ]
            tag_on_failure => [ "_grok_postfix_anvil_nomatch" ]
            add_tag        => [ "_grok_postfix_success" ]
        }
    } else if [program] =~ /^postfix.*\/bounce$/ {
        grok {
            patterns_dir   => ["/etc/logstash/patterns.d"]
            match          => [ "message", "%{POSTFIX_BOUNCE}" ]
            tag_on_failure => [ "_grok_postfix_bounce_nomatch" ]
            add_tag        => [ "_grok_postfix_success" ]
        }
    } else if [program] =~ /^postfix.*\/cleanup$/ {
        grok {
            patterns_dir   => ["/etc/logstash/patterns.d"]
            match          => [ "message", "%{POSTFIX_CLEANUP}"
]
            tag_on_failure => [ "_grok_postfix_cleanup_nomatch"
]
            add_tag        => [ "_grok_postfix_success" ]
        }
    } else if [program] =~ /^postfix.*\/dnsblog$/ {
        grok {
            patterns_dir   => ["/etc/logstash/patterns.d"]
```

```
            match            => [ "message", "%{POSTFIX_DNSBLOG}"
]
            tag_on_failure => [ "_grok_postfix_dnsblog_nomatch"
]
            add_tag        => [ "_grok_postfix_success" ]
        }
    } else if [program] =~ /^postfix.*\/local$/ {
        grok {
            patterns_dir   => ["/etc/logstash/patterns.d"]
            match          => [ "message", "%{POSTFIX_LOCAL}" ]
            tag_on_failure => [ "_grok_postfix_local_nomatch" ]
            add_tag        => [ "_grok_postfix_success" ]
        }
    } else if [program] =~ /^postfix.*\/master$/ {
        grok {
            patterns_dir   => ["/etc/logstash/patterns.d"]
            match          => [ "message", "%{POSTFIX_MASTER}" ]
            tag_on_failure => [ "_grok_postfix_master_nomatch" ]
            add_tag        => [ "_grok_postfix_success" ]
        }
    } else if [program] =~ /^postfix.*\/pickup$/ {
        grok {
            patterns_dir   => ["/etc/logstash/patterns.d"]
            match          => [ "message", "%{POSTFIX_PICKUP}" ]
            tag_on_failure => [ "_grok_postfix_pickup_nomatch" ]
            add_tag        => [ "_grok_postfix_success" ]
        }
    } else if [program] =~ /^postfix.*\/pipe$/ {
        grok {
            patterns_dir   => ["/etc/logstash/patterns.d"]
            match          => [ "message", "%{POSTFIX_PIPE}" ]
            tag_on_failure => [ "_grok_postfix_pipe_nomatch" ]
            add_tag        => [ "_grok_postfix_success" ]
        }
    } else if [program] =~ /^postfix.*\/postdrop$/ {
        grok {
            patterns_dir   => ["/etc/logstash/patterns.d"]
            match          => [ "message", "%{POSTFIX_POSTDROP}"
 ]
            tag_on_failure => [ "_grok_postfix_postdrop_nomatch"
```

```
 ]
                add_tag         => [ "_grok_postfix_success" ]
            }
        } else if [program] =~ /^postfix.*\/postscreen$/ {
            grok {
                patterns_dir   => ["/etc/logstash/patterns.d"]
                match          => [ "message", "%{POSTFIX_POSTSCREEN
}" ]
                tag_on_failure => [ "_grok_postfix_postscreen_nomatc
h" ]
                add_tag         => [ "_grok_postfix_success" ]
            }
        } else if [program] =~ /^postfix.*\/qmgr$/ {
            grok {
                patterns_dir   => ["/etc/logstash/patterns.d"]
                match          => [ "message", "%{POSTFIX_QMGR}" ]
                tag_on_failure => [ "_grok_postfix_qmgr_nomatch" ]
                add_tag         => [ "_grok_postfix_success" ]
            }
        } else if [program] =~ /^postfix.*\/scache$/ {
            grok {
                patterns_dir   => ["/etc/logstash/patterns.d"]
                match          => [ "message", "%{POSTFIX_SCACHE}" ]
                tag_on_failure => [ "_grok_postfix_scache_nomatch" ]
                add_tag         => [ "_grok_postfix_success" ]
            }
        } else if [program] =~ /^postfix.*\/sendmail$/ {
            grok {
                patterns_dir   => ["/etc/logstash/patterns.d"]
                match          => [ "message", "%{POSTFIX_SENDMAIL}"
 ]
                tag_on_failure => [ "_grok_postfix_sendmail_nomatch"
 ]
                add_tag         => [ "_grok_postfix_success" ]
            }
        } else if [program] =~ /^postfix.*\/smtp$/ {
            grok {
                patterns_dir   => ["/etc/logstash/patterns.d"]
                match          => [ "message", "%{POSTFIX_SMTP}" ]
                tag_on_failure => [ "_grok_postfix_smtp_nomatch" ]
```

```
            add_tag         => [ "_grok_postfix_success" ]
        }
    } else if [program] =~ /^postfix.*\/lmtp$/ {
        grok {
            patterns_dir   => ["/etc/logstash/patterns.d"]
            match          => [ "message", "%{POSTFIX_LMTP}" ]
            tag_on_failure => [ "_grok_postfix_lmtp_nomatch" ]
            add_tag        => [ "_grok_postfix_success" ]
        }
    } else if [program] =~ /^postfix.*\/smtpd$/ {
        grok {
            patterns_dir   => ["/etc/logstash/patterns.d"]
            match          => [ "message", "%{POSTFIX_SMTPD}" ]
            tag_on_failure => [ "_grok_postfix_smtpd_nomatch" ]
            add_tag        => [ "_grok_postfix_success" ]
        }
    } else if [program] =~ /^postfix.*\/tlsmgr$/ {
        grok {
            patterns_dir   => ["/etc/logstash/patterns.d"]
            match          => [ "message", "%{POSTFIX_TLSMGR}" ]
            tag_on_failure => [ "_grok_postfix_tlsmgr_nomatch" ]
            add_tag        => [ "_grok_postfix_success" ]
        }
    } else if [program] =~ /^postfix.*\/tlsproxy$/ {
        grok {
            patterns_dir   => ["/etc/logstash/patterns.d"]
            match          => [ "message", "%{POSTFIX_TLSPROXY}"
 ]
            tag_on_failure => [ "_grok_postfix_tlsproxy_nomatch"
 ]
            add_tag        => [ "_grok_postfix_success" ]
        }
    } else if [program] =~ /^postfix.*\/trivial-rewrite$/ {
        grok {
            patterns_dir   => ["/etc/logstash/patterns.d"]
            match          => [ "message", "%{POSTFIX_TRIVIAL_RE
WRITE}" ]
            tag_on_failure => [ "_grok_postfix_trivial_rewrite_n
omatch" ]
            add_tag        => [ "_grok_postfix_success" ]
```

```
        }
    } else if [program] =~ /^postfix.*\/discard$/ {
        grok {
            patterns_dir   => ["/etc/logstash/patterns.d"]
            match          => [ "message", "%{POSTFIX_DISCARD}"
]
            tag_on_failure => [ "_grok_postfix_discard_nomatch"
]
            add_tag        => [ "_grok_postfix_success" ]
        }
    }

    # process key-value data is it exists
    if [postfix_keyvalue_data] {
        kv {
            source       => "postfix_keyvalue_data"
            trim         => "<>,"
            prefix       => "postfix_"
            remove_field => [ "postfix_keyvalue_data" ]
        }

        # some post processing of key-value data
        if [postfix_client] {
            grok {
                patterns_dir   => ["/etc/logstash/patterns.d"]
                match          => ["postfix_client", "%{POSTFIX_
CLIENT_INFO}"]
                tag_on_failure => [ "_grok_kv_postfix_client_nom
atch" ]
                remove_field   => [ "postfix_client" ]
            }
        }
        if [postfix_relay] {
            grok {
                patterns_dir   => ["/etc/logstash/patterns.d"]
                match          => ["postfix_relay", "%{POSTFIX_R
ELAY_INFO}"]
                tag_on_failure => [ "_grok_kv_postfix_relay_noma
tch" ]
                remove_field   => [ "postfix_relay" ]
```

```
            }
        }
        if [postfix_delays] {
            grok {
                patterns_dir   => ["/etc/logstash/patterns.d"]
                match          => ["postfix_delays", "%{POSTFIX_
DELAYS}"]
                tag_on_failure => [ "_grok_kv_postfix_delays_nom
atch" ]
                remove_field   => [ "postfix_delays" ]
            }
        }
    }

    # Do some data type conversions
    mutate {
        convert => [
            # list of integer fields
            "postfix_anvil_cache_size", "integer",
            "postfix_anvil_conn_count", "integer",
            "postfix_anvil_conn_rate", "integer",
            "postfix_client_port", "integer",
            "postfix_nrcpt", "integer",
            "postfix_postscreen_cache_dropped", "integer",
            "postfix_postscreen_cache_retained", "integer",
            "postfix_postscreen_dnsbl_rank", "integer",
            "postfix_relay_port", "integer",
            "postfix_server_port", "integer",
            "postfix_size", "integer",
            "postfix_status_code", "integer",
            "postfix_termination_signal", "integer",
            "postfix_uid", "integer",

            # list of float fields
            "postfix_delay", "float",
            "postfix_delay_before_qmgr", "float",
            "postfix_delay_conn_setup", "float",
            "postfix_delay_in_qmgr", "float",
            "postfix_delay_transmission", "float",
            "postfix_postscreen_violation_time", "float"
```

```
        ]
    }
}
```

配置中使用了一系列自定义 grok 正则，全部内容如下所示。保存成 `/etc/logstash/patterns.d/` 下一个文本文件即可。

```
# common postfix patterns
POSTFIX_QUEUEID ([0-9A-F]{6,}|[0-9a-zA-Z]{15,}|NOQUEUE)
POSTFIX_CLIENT_INFO %{HOST:postfix_client_hostname}?\[%{IP:postfix_client_ip}\](:%{INT:postfix_client_port})?
POSTFIX_RELAY_INFO %{HOST:postfix_relay_hostname}?\[(%{IP:postfix_relay_ip}|%{DATA:postfix_relay_service})\](:%{INT:postfix_relay_port})?|%{WORD:postfix_relay_service}
POSTFIX_SMTP_STAGE (CONNECT|HELO|EHLO|STARTTLS|AUTH|MAIL|RCPT|DATA|RSET|UNKNOWN|END-OF-MESSAGE|VRFY|\.)
POSTFIX_ACTION (reject|defer|accept|header-redirect)
POSTFIX_STATUS_CODE \d{3}
POSTFIX_STATUS_CODE_ENHANCED \d\.\d\.\d
POSTFIX_DNSBL_MESSAGE Service unavailable; .* \[%{GREEDYDATA:postfix_status_data}\] %{GREEDYDATA:postfix_status_message};
POSTFIX_PS_ACCESS_ACTION (DISCONNECT|BLACKLISTED|WHITELISTED|WHITELIST VETO|PASS NEW|PASS OLD)
POSTFIX_PS_VIOLATION (BARE NEWLINE|COMMAND (TIME|COUNT|LENGTH) LIMIT|COMMAND PIPELINING|DNSBL|HANGUP|NON-SMTP COMMAND|PREGREET)
POSTFIX_TIME_UNIT %{NUMBER}[smhd]
POSTFIX_KEYVALUE %{POSTFIX_QUEUEID:postfix_queueid}: %{GREEDYDATA:postfix_keyvalue_data}
POSTFIX_WARNING (warning|fatal): %{GREEDYDATA:postfix_warning}
POSTFIX_TLSCONN (Anonymous|Trusted|Untrusted|Verified) TLS connection established (to %{POSTFIX_RELAY_INFO}|from %{POSTFIX_CLIENT_INFO}): %{DATA:postfix_tls_version} with cipher %{DATA:postfix_tls_cipher} \(%{DATA:postfix_tls_cipher_size} bits\)
POSTFIX_DELAYS %{NUMBER:postfix_delay_before_qmgr}/%{NUMBER:postfix_delay_in_qmgr}/%{NUMBER:postfix_delay_conn_setup}/%{NUMBER:postfix_delay_transmission}
POSTFIX_LOSTCONN (lost connection|timeout|Connection timed out)
POSTFIX_PROXY_MESSAGE (%{POSTFIX_STATUS_CODE:postfix_proxy_status_code} )?(%{POSTFIX_STATUS_CODE_ENHANCED:postfix_proxy_status_c
```

```
ode_enhanced})?.*

# helper patterns
GREEDYDATA_NO_COLON [^:]*

# smtpd patterns
POSTFIX_SMTPD_CONNECT connect from %{POSTFIX_CLIENT_INFO}
POSTFIX_SMTPD_DISCONNECT disconnect from %{POSTFIX_CLIENT_INFO}
POSTFIX_SMTPD_LOSTCONN (%{POSTFIX_LOSTCONN:postfix_smtpd_lostcon
n_data} after %{POSTFIX_SMTP_STAGE:postfix_smtp_stage}( \(%{INT}
 bytes\))? from %{POSTFIX_CLIENT_INFO}|%{GREEDYDATA:postfix_acti
on} from %{POSTFIX_CLIENT_INFO}: %{POSTFIX_LOSTCONN:postfix_smtp
d_lostconn_data})
POSTFIX_SMTPD_NOQUEUE NOQUEUE: %{POSTFIX_ACTION:postfix_action}:
 %{POSTFIX_SMTP_STAGE:postfix_smtp_stage} from %{POSTFIX_CLIENT_
INFO}: %{POSTFIX_STATUS_CODE:postfix_status_code} %{POSTFIX_STAT
US_CODE_ENHANCED:postfix_status_code_enhanced}( <%{DATA:postfix_
status_data}>:)? (%{POSTFIX_DNSBL_MESSAGE}|%{GREEDYDATA:postfix_
status_message};) %{GREEDYDATA:postfix_keyvalue_data}
POSTFIX_SMTPD_PIPELINING improper command pipelining after %{POS
TFIX_SMTP_STAGE:postfix_smtp_stage} from %{POSTFIX_CLIENT_INFO}:
POSTFIX_SMTPD_PROXY proxy-%{POSTFIX_ACTION:postfix_proxy_result}
: (%{POSTFIX_SMTP_STAGE:postfix_proxy_smtp_stage}): %{POSTFIX_PR
OXY_MESSAGE:postfix_proxy_message}; %{GREEDYDATA:postfix_keyvalu
e_data}

# cleanup patterns
POSTFIX_CLEANUP_MILTER %{POSTFIX_QUEUEID:postfix_queueid}: milte
r-%{POSTFIX_ACTION:postfix_milter_result}: %{GREEDYDATA:postfix_
milter_message}; %{GREEDYDATA_NO_COLON:postfix_keyvalue_data}(:
%{GREEDYDATA:postfix_milter_data})?

# qmgr patterns
POSTFIX_QMGR_REMOVED %{POSTFIX_QUEUEID:postfix_queueid}: removed
POSTFIX_QMGR_ACTIVE %{POSTFIX_QUEUEID:postfix_queueid}: %{GREEDY
DATA:postfix_keyvalue_data} \(queue active\)

# pipe patterns
POSTFIX_PIPE_DELIVERED %{POSTFIX_QUEUEID:postfix_queueid}: %{GRE
EDYDATA:postfix_keyvalue_data} \(delivered via %{WORD:postfix_pi
```

```
pe_service} service\)
POSTFIX_PIPE_FORWARD %{POSTFIX_QUEUEID:postfix_queueid}: %{GREED
YDATA:postfix_keyvalue_data} \(mail forwarding loop for %{GREEDY
DATA:postfix_to}\)

# postscreen patterns
POSTFIX_PS_CONNECT CONNECT from %{POSTFIX_CLIENT_INFO} to \[%{IP
:postfix_server_ip}\]:%{INT:postfix_server_port}
POSTFIX_PS_ACCESS %{POSTFIX_PS_ACCESS_ACTION:postfix_postscreen_
access} %{POSTFIX_CLIENT_INFO}
POSTFIX_PS_NOQUEUE %{POSTFIX_SMTPD_NOQUEUE}
POSTFIX_PS_TOOBUSY NOQUEUE: reject: CONNECT from %{POSTFIX_CLIEN
T_INFO}: %{GREEDYDATA:postfix_postscreen_toobusy_data}
POSTFIX_PS_DNSBL %{POSTFIX_PS_VIOLATION:postfix_postscreen_viola
tion} rank %{INT:postfix_postscreen_dnsbl_rank} for %{POSTFIX_CL
IENT_INFO}
POSTFIX_PS_CACHE cache %{DATA} full cleanup: retained=%{NUMBER:p
ostfix_postscreen_cache_retained} dropped=%{NUMBER:postfix_posts
creen_cache_dropped} entries
POSTFIX_PS_VIOLATIONS %{POSTFIX_PS_VIOLATION:postfix_postscreen_
violation}( %{INT})?( after %{NUMBER:postfix_postscreen_violatio
n_time})? from %{POSTFIX_CLIENT_INFO}( after %{POSTFIX_SMTP_STAG
E:postfix_smtp_stage})?

# dnsblog patterns
POSTFIX_DNSBLOG_LISTING addr %{IP:postfix_client_ip} listed by d
omain %{HOST:postfix_dnsbl_domain} as %{IP:postfix_dnsbl_result}

# tlsproxy patterns
POSTFIX_TLSPROXY_CONN (DIS)?CONNECT( from)? %{POSTFIX_CLIENT_INF
O}

# anvil patterns
POSTFIX_ANVIL_CONN_RATE statistics: max connection rate %{NUMBER
:postfix_anvil_conn_rate}/%{POSTFIX_TIME_UNIT:postfix_anvil_conn
_period} for \(%{DATA:postfix_service}:%{IP:postfix_client_ip}\)
 at %{SYSLOGTIMESTAMP:postfix_anvil_timestamp}
POSTFIX_ANVIL_CONN_CACHE statistics: max cache size %{NUMBER:pos
tfix_anvil_cache_size} at %{SYSLOGTIMESTAMP:postfix_anvil_timest
amp}
```

```
POSTFIX_ANVIL_CONN_COUNT statistics: max connection count %{NUMB
ER:postfix_anvil_conn_count} for \(%{DATA:postfix_service}:%{IP:
postfix_client_ip}\) at %{SYSLOGTIMESTAMP:postfix_anvil_timestam
p}

# smtp patterns
POSTFIX_SMTP_DELIVERY %{POSTFIX_KEYVALUE} status=%{WORD:postfix_
status}( \(%{GREEDYDATA:postfix_smtp_response}\))?
POSTFIX_SMTP_CONNERR connect to %{POSTFIX_RELAY_INFO}: (Connecti
on timed out|No route to host|Connection refused)
POSTFIX_SMTP_LOSTCONN %{POSTFIX_QUEUEID:postfix_queueid}: %{POST
FIX_LOSTCONN} with %{POSTFIX_RELAY_INFO}

# master patterns
POSTFIX_MASTER_START (daemon started|reload) -- version %{DATA:p
ostfix_version}, configuration %{PATH:postfix_config_path}
POSTFIX_MASTER_EXIT terminating on signal %{INT:postfix_terminat
ion_signal}

# bounce patterns
POSTFIX_BOUNCE_NOTIFICATION %{POSTFIX_QUEUEID:postfix_queueid}:
sender (non-delivery|delivery status|delay) notification: %{POST
FIX_QUEUEID:postfix_bounce_queueid}

# scache patterns
POSTFIX_SCACHE_LOOKUPS statistics: (address|domain) lookup hits=
%{INT:postfix_scache_hits} miss=%{INT:postfix_scache_miss} succe
ss=%{INT:postfix_scache_success}
POSTFIX_SCACHE_SIMULTANEOUS statistics: max simultaneous domains
=%{INT:postfix_scache_domains} addresses=%{INT:postfix_scache_ad
dresses} connection=%{INT:postfix_scache_connection}
POSTFIX_SCACHE_TIMESTAMP statistics: start interval %{SYSLOGTIME
STAMP:postfix_scache_timestamp}

# aggregate all patterns
POSTFIX_SMTPD %{POSTFIX_SMTPD_CONNECT}|%{POSTFIX_SMTPD_DISCONNEC
T}|%{POSTFIX_SMTPD_LOSTCONN}|%{POSTFIX_SMTPD_NOQUEUE}|%{POSTFIX_
SMTPD_PIPELINING}|%{POSTFIX_TLSCONN}|%{POSTFIX_WARNING}|%{POSTFI
X_SMTPD_PROXY}|%{POSTFIX_KEYVALUE}
POSTFIX_CLEANUP %{POSTFIX_CLEANUP_MILTER}|%{POSTFIX_WARNING}|%{P
```

```
OSTFIX_KEYVALUE}
POSTFIX_QMGR %{POSTFIX_QMGR_REMOVED}|%{POSTFIX_QMGR_ACTIVE}|%{PO
STFIX_WARNING}
POSTFIX_PIPE %{POSTFIX_PIPE_DELIVERED}|%{POSTFIX_PIPE_FORWARD}
POSTFIX_POSTSCREEN %{POSTFIX_PS_CONNECT}|%{POSTFIX_PS_ACCESS}|%{
POSTFIX_PS_NOQUEUE}|%{POSTFIX_PS_TOOBUSY}|%{POSTFIX_PS_CACHE}|%{
POSTFIX_PS_DNSBL}|%{POSTFIX_PS_VIOLATIONS}|%{POSTFIX_WARNING}
POSTFIX_DNSBLOG %{POSTFIX_DNSBLOG_LISTING}
POSTFIX_ANVIL %{POSTFIX_ANVIL_CONN_RATE}|%{POSTFIX_ANVIL_CONN_CA
CHE}|%{POSTFIX_ANVIL_CONN_COUNT}
POSTFIX_SMTP %{POSTFIX_SMTP_DELIVERY}|%{POSTFIX_SMTP_CONNERR}|%{
POSTFIX_SMTP_LOSTCONN}|%{POSTFIX_TLSCONN}|%{POSTFIX_WARNING}
POSTFIX_DISCARD %{POSTFIX_KEYVALUE} status=%{WORD:postfix_status
}
POSTFIX_LMTP %{POSTFIX_SMTP}
POSTFIX_PICKUP %{POSTFIX_KEYVALUE}
POSTFIX_TLSPROXY %{POSTFIX_TLSPROXY_CONN}
POSTFIX_MASTER %{POSTFIX_MASTER_START}|%{POSTFIX_MASTER_EXIT}
POSTFIX_BOUNCE %{POSTFIX_BOUNCE_NOTIFICATION}
POSTFIX_SENDMAIL %{POSTFIX_WARNING}
POSTFIX_POSTDROP %{POSTFIX_WARNING}
POSTFIX_SCACHE %{POSTFIX_SCACHE_LOOKUPS}|%{POSTFIX_SCACHE_SIMULT
ANEOUS}|%{POSTFIX_SCACHE_TIMESTAMP}
POSTFIX_TRIVIAL_REWRITE %{POSTFIX_WARNING}
POSTFIX_TLSMGR %{POSTFIX_WARNING}
POSTFIX_LOCAL %{POSTFIX_KEYVALUE}
```

ossec

本节作者：林鹏

配置OSSEC SYSLOG 输出 （所有agent）

1. 编辑ossec.conf 文件（默认为/var/ossec/etc/ossec.conf）
2. 在ossec.conf中添加下列内容（10.0.0.1 为 接收syslog 的服务器）

```
<syslog_output>
   <server>10.0.0.1</server>
   <port>9000</port>
   <format>default</format>
</syslog_output>
```

1. 开启OSSEC允许syslog输出功能

```
/var/ossec/bin/ossec-control enable client-syslog
```

1. 重启 OSSEC服务

   ```
   /var/ossec/bin/ossec-control start
   ```

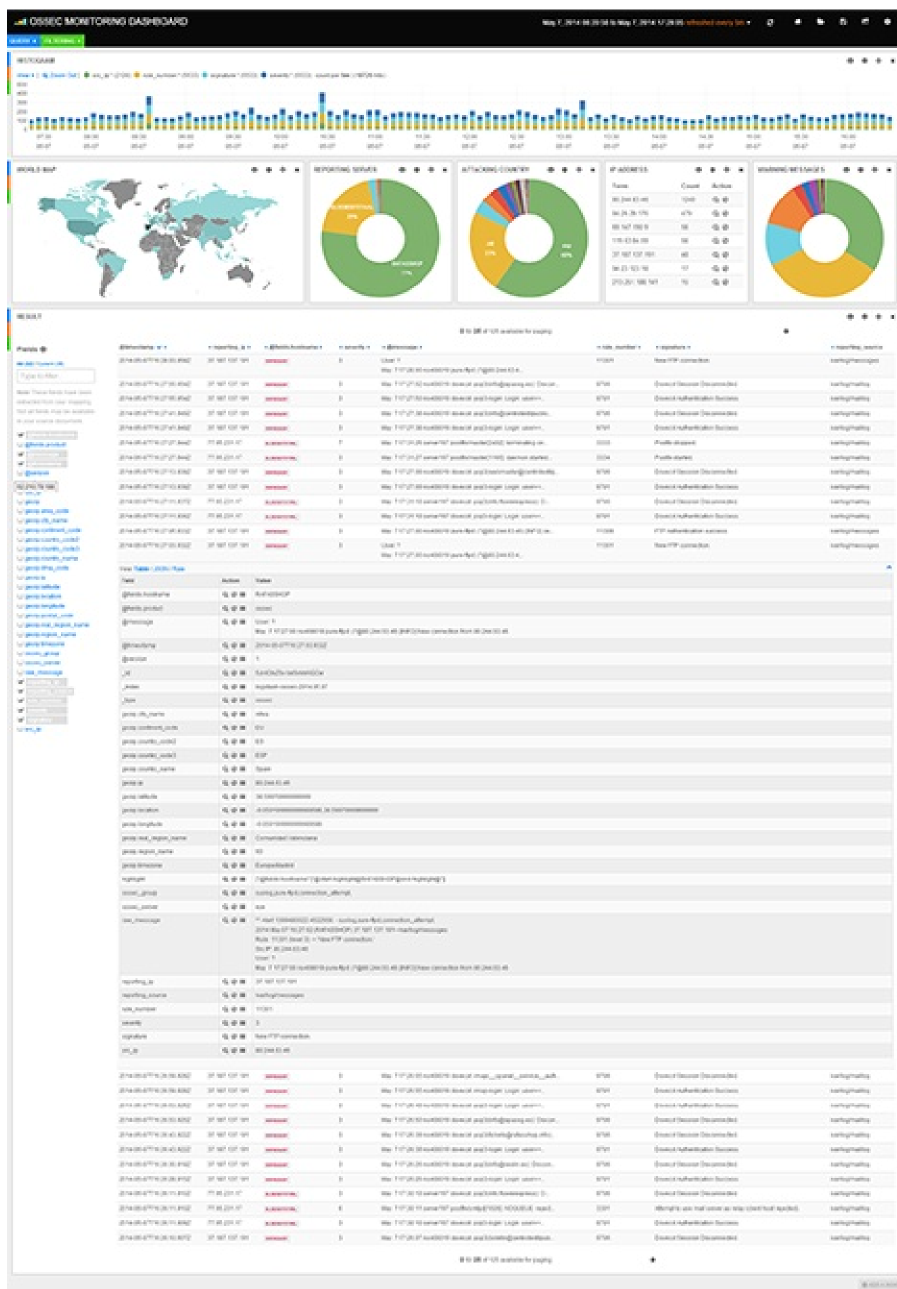
配置LOGSTASH

1. 在logstash 中 配置文件中增加(或新建)如下内容：（假设10.0.0.1 为ES服务器,假设文件名为logstash-ossec.conf ）

```
input {
    udp {
        port => 9000
        type => "syslog"
    }
}
filter {
    if [type] == "syslog" {
        grok {
            match => { "message" => "%{SYSLOGTIMESTAMP:syslog_ti
mestamp} %{SYSLOGHOST:syslog_host} %{DATA:syslog_program}: Alert
 Level: %{BASE10NUM:Alert_Level}; Rule: %{BASE10NUM:Rule} - %{GR
EEDYDATA:Description}; Location: %{GREEDYDATA:Details}" }
            add_field => [ "ossec_server", "%{host}" ]
        }
        mutate {
            remove_field => [ "syslog_hostname", "syslog_message
", "syslog_pid", "message", "@version", "type", "host" ]
        }
    }
}
output {
    elasticsearch_http {
        host => "10.0.0.1"
    }
}
```

推荐 Kibana dashboard

社区已经有人根据 ossec 的常见需求，制作有 dashboard 可以直接从 Kibana3 页面加载使用。



dashboard 的 JSON 文件见：https://github.com/magenx/Logstash/raw/master/kibana/kibana_dashboard.json

加载方式，请阅读本书稍后 Kibana 章节相关内容。

Windows Event Log

前面说过如何在 windows 上利用 nxlog 传输日志数据。事实上，对于 windows 本身，也有类似 syslog 的设计，叫 eventlog。本节介绍如何处理 windows eventlog。

采集端配置

logstash 配置

```
input {
    eventlog {
        #logfile =>  ["Application", "Security", "System"]
        logfile =>  ["Security"]
        type => "winevent"
        tags => [ "caen" ]
    }
}
```

nxlog 配置

```
## This is a sample configuration file. See the nxlog reference
manual about the
## configuration options. It should be installed locally and is
also available
## online at http://nxlog.org/nxlog-docs/en/nxlog-reference-manu
al.html

## Please set the ROOT to the folder your nxlog was installed in
to,
## otherwise it will not start.

#define ROOT C:\Program Files\nxlog
define ROOT C:\Program Files (x86)\nxlog
```

```
Moduledir %ROOT%\modules
CacheDir %ROOT%\data
Pidfile %ROOT%\data\nxlog.pid
SpoolDir %ROOT%\data
LogFile %ROOT%\data\nxlog.log

<Extension json>
    Module    xm_json
</Extension>

<Input in>
    Module      im_msvistalog
# For windows 2003 and earlier use the following:
#   Module      im_mseventlog
    Exec    to_json();
</Input>

<Output out>
    Module      om_tcp
    Host        10.66.66.66
    Port        5140
</Output>

<Route 1>
    Path        in => out
</Route>
```

Logstash 解析配置

```
input {
  tcp {
    codec => "json"
    port => 5140
    tags => ["windows","nxlog"]
    type => "nxlog-json"
  }
} # end input
```

```
filter {
  if [type] == "nxlog-json" {
    date {
      match => ["[EventTime]", "YYYY-MM-dd HH:mm:ss"]
      timezone => "Europe/London"
    }
    mutate {
        rename => [ "AccountName", "user" ]
        rename => [ "AccountType", "[eventlog][account_type]" ]
        rename => [ "ActivityId", "[eventlog][activity_id]" ]
        rename => [ "Address", "ip6" ]
        rename => [ "ApplicationPath", "[eventlog][application_p
ath]" ]
        rename => [ "AuthenticationPackageName", "[eventlog][aut
hentication_package_name]" ]
        rename => [ "Category", "[eventlog][category]" ]
        rename => [ "Channel", "[eventlog][channel]" ]
        rename => [ "Domain", "domain" ]
        rename => [ "EventID", "[eventlog][event_id]" ]
        rename => [ "EventType", "[eventlog][event_type]" ]
        rename => [ "File", "[eventlog][file_path]" ]
        rename => [ "Guid", "[eventlog][guid]" ]
        rename => [ "Hostname", "hostname" ]
        rename => [ "Interface", "[eventlog][interface]" ]
        rename => [ "InterfaceGuid", "[eventlog][interface_guid]
" ]
        rename => [ "InterfaceName", "[eventlog][interface_name]
" ]
        rename => [ "IpAddress", "ip" ]
        rename => [ "IpPort", "port" ]
        rename => [ "Key", "[eventlog][key]" ]
        rename => [ "LogonGuid", "[eventlog][logon_guid]" ]
        rename => [ "Message", "message" ]
        rename => [ "ModifyingUser", "[eventlog][modifying_user]
" ]
        rename => [ "NewProfile", "[eventlog][new_profile]" ]
        rename => [ "OldProfile", "[eventlog][old_profile]" ]
        rename => [ "Port", "port" ]
        rename => [ "PrivilegeList", "[eventlog][privilege_list]
```

```
" ]
        rename => [ "ProcessID", "pid" ]
        rename => [ "ProcessName", "[eventlog][process_name]" ]
        rename => [ "ProviderGuid", "[eventlog][provider_guid]"
]
        rename => [ "ReasonCode", "[eventlog][reason_code]" ]
        rename => [ "RecordNumber", "[eventlog][record_number]"
]
        rename => [ "ScenarioId", "[eventlog][scenario_id]" ]
        rename => [ "Severity", "level" ]
        rename => [ "SeverityValue", "[eventlog][severity_code]"
 ]
        rename => [ "SourceModuleName", "nxlog_input" ]
        rename => [ "SourceName", "[eventlog][program]" ]
        rename => [ "SubjectDomainName", "[eventlog][subject_dom
ain_name]" ]
        rename => [ "SubjectLogonId", "[eventlog][subject_logoni
d]" ]
        rename => [ "SubjectUserName", "[eventlog][subject_user_
name]" ]
        rename => [ "SubjectUserSid", "[eventlog][subject_user_s
id]" ]
        rename => [ "System", "[eventlog][system]" ]
        rename => [ "TargetDomainName", "[eventlog][target_domai
n_name]" ]
        rename => [ "TargetLogonId", "[eventlog][target_logonid]
" ]
        rename => [ "TargetUserName", "[eventlog][target_user_na
me]" ]
        rename => [ "TargetUserSid", "[eventlog][target_user_sid
]" ]
        rename => [ "ThreadID", "thread" ]
    }
    mutate {
        remove_field => [
                    "CurrentOrNextState",
                    "Description",
                    "EventReceivedTime",
                    "EventTime",
                    "EventTimeWritten",
```

```
                    "IPVersion",
                    "KeyLength",
                    "Keywords",
                    "LmPackageName",
                    "LogonProcessName",
                    "LogonType",
                    "Name",
                    "Opcode",
                    "OpcodeValue",
                    "PolicyProcessingMode",
                    "Protocol",
                    "ProtocolType",
                    "SourceModuleType",
                    "State",
                    "Task",
                    "TransmittedServices",
                    "Type",
                    "UserID",
                    "Version"
                    ]
    }
  }

}
```

Java 日志

之前在 codec 章节，曾经提到过，对 Java 日志，除了使用 multiline 做多行日志合并以外，还可以直接通过 log4j 写入到 logstash 里。本节就讲述如何在 Java 应用环境做到这点。

Log4J

首先，需要配置 Java 应用的 Log4J 设置，启动一个内置的 `SocketAppender` 。修改应用的 `log4j.xml` 配置文件，添加如下配置段：

```
<appender name="LOGSTASH" class="org.apache.log4j.net.SocketAppe
nder">
    <param name="RemoteHost" value="logstash_hostname" />
    <param name="ReconnectionDelay" value="60000" />
    <param name="LocationInfo" value="true" />
    <param name="Threshold" value="DEBUG" />
</appender>
```

然后把这个新定义的 appender 对象加入到 root logger 里，可以跟其他已有 logger 共存：

```
<root>
    <level value="INFO"/>
    <appender-ref ref="OTHERPLACE"/>
    <appender-ref ref="LOGSTASH"/>
</root>
```

如果是 `log4j.properties` 配置文件，则对应配置如下：

```
log4j.rootLogger=DEBUG, logstash

###SocketAppender###
log4j.appender.logstash=org.apache.log4j.net.SocketAppender
log4j.appender.logstash.Port=4560
log4j.appender.logstash.RemoteHost=logstash_hostname
log4j.appender.logstash.ReconnectionDelay=60000
log4j.appender.logstash.LocationInfo=true
```

Log4J 会持续尝试连接你配置的 *logstash_hostname* 这个地址，建立连接后，即开始发送日志数据。

Logstash

Java 应用端的配置完成以后，开始设置 Logstash 的接收端。配置如下所示。其中 4560 端口是 Log4J SocketAppender 的默认对端端口。

```
input {
  log4j {
    type => "log4j-json"
    port => 4560
  }
}
```

异常堆栈测试验证

运行起来 logstash 后，编写如下一个简单 log4j 程序：

```
import org.apache.log4j.Logger;
public class HelloExample{
        final static Logger logger = Logger.getLogger(HelloExamp
le.class);
        public static void main(String[] args) {
                HelloExample obj = new HelloExample();
                try{
                        obj.divide();
                }catch(ArithmeticException ex){
                        logger.error("Sorry, something wrong!",
ex);
                }
        }
        private void divide(){
                int i = 10 /0;
        }
}
```

编译运行：

```
# javac -cp ./logstash-1.5.0.rc2/vendor/bundle/jruby/1.9/gems/lo
gstash-input-log4j-0.1.3-java/lib/log4j/log4j/1.2.17/log4j-1.2.1
7.jar HelloExample.java
# java -cp .:./logstash-1.5.0.rc2/vendor/bundle/jruby/1.9/gems/l
ogstash-input-log4j-0.1.3-java/lib/log4j/log4j/1.2.17/log4j-1.2.
17.jar HelloExample
```

即可在 logstash 的终端输出看到如下事件记录：

```
{
        "message" => "Sorry, something wrong!",
       "@version" => "1",
     "@timestamp" => "2015-07-02T13:24:45.727Z",
           "type" => "log4j-json",
           "host" => "127.0.0.1:52420",
           "path" => "HelloExample",
       "priority" => "ERROR",
    "logger_name" => "HelloExample",
         "thread" => "main",
          "class" => "HelloExample",
           "file" => "HelloExample.java:9",
         "method" => "main",
    "stack_trace" => "java.lang.ArithmeticException: / by zero\n
\tat HelloExample.divide(HelloExample.java:13)\n\tat HelloExampl
e.main(HelloExample.java:7)"
}
```

可以看到，异常堆栈直接就记录在单行内了。

JSON Event layout

如果无法采用 socketappender ，必须使用文件方式的，其实 Log4J 有一个 layout 特性，用来控制日志输出的格式。和 Nginx 日志自己拼接 JSON 输出类似，也可以通过 layout 功能，记录成 JSON 格式。推荐使用：https://github.com/logstash/log4j-jsonevent-layout

MySQL慢查询日志

MySQL 有多种日志可以记录，常见的有 error log、slow log、general log、bin log 等。其中 slow log 作为性能监控和优化的入手点，最为首要。本节即讨论如何用 logstash 处理 slow log。至于 general log，格式处理基本类似，不过由于 general 量级比 slow 大得多，推荐采用 packetbeat 协议解析的方式更高效的完成这项工作。相关内容阅读本书稍后章节。

MySQL slow log 的 logstash 处理配置示例如下：

```
input {
  file {
    type => "mysql-slow"
    path => "/var/log/mysql/mysql-slow.log"
    codec => multiline {
      pattern => "^# User@Host:"
      negate => true
      what => "previous"
    }
  }
}

filter {
  # drop sleep events
  grok {
    match => { "message" => "SELECT SLEEP" }
    add_tag => [ "sleep_drop" ]
    tag_on_failure => [] # prevent default _grokparsefailure tag
 on real records
  }
  if "sleep_drop" in [tags] {
    drop {}
  }
  grok {
    match => [ "message", "(?m)^# User@Host: %{USER:user}\[[^\]]
+\] @ (?:(?<clienthost>\S*) )?\[(?:%{IP:clientip})?\]\s*# Query_
time: %{NUMBER:query_time:float}\s+Lock_time: %{NUMBER:lock_time
:float}\s+Rows_sent: %{NUMBER:rows_sent:int}\s+Rows_examined: %{
NUMBER:rows_examined:int}\s*(?:use %{DATA:database};\s*)?SET tim
estamp=%{NUMBER:timestamp};\s*(?<query>(?<action>\w+)\s+.*)\n# T
ime:.*$" ]
  }
  date {
    match => [ "timestamp", "UNIX" ]
    remove_field => [ "timestamp" ]
  }
}
```

运行该配置，logstash 即可将多行的 MySQL slow log 处理成如下事件：

```
{
       "@timestamp" => "2014-03-04T19:59:06.000Z",
          "message" => "# User@Host: logstash[logstash] @ localh
ost [127.0.0.1]\n# Query_time: 5.310431  Lock_time: 0.029219 Row
s_sent: 1  Rows_examined: 24575727\nSET timestamp=1393963146;\ns
elect count(*) from node join variable order by rand();\n# Time:
 140304 19:59:14",
         "@version" => "1",
             "tags" => [
        [0] "multiline"
    ],
             "type" => "mysql-slow",
             "host" => "raochenlindeMacBook-Air.local",
             "path" => "/var/log/mysql/mysql-slow.log",
             "user" => "logstash",
       "clienthost" => "localhost",
         "clientip" => "127.0.0.1",
       "query_time" => 5.310431,
        "lock_time" => 0.029219,
        "rows_sent" => 1,
    "rows_examined" => 24575727,
            "query" => "select count(*) from node join variable
order by rand();",
           "action" => "select"
}
```

性能与测试

任何软件都需要掌握其性能瓶颈，以及线上运行时的性能状态。Logstash 也不例外。

长久以来，Logstash 在这方面一直处于比较黑盒的状态。因为其内部队列使用的是标准的 stud 库，并非自己实现，在 Logstash 本身源代码里是找不出来什么问题的。我们只能按照其 pipeline 原理，总结出来一些模拟检测的手段。

在 Logstash-5.0.0 中，一大改进就是学习 Elasticsearch 的方式，通过 API 提供了一部分运行性能指标！本节就会介绍这方面的内容。同时，作为极限压测的方式，依然会介绍一些模拟数据的生成和 JVM 指标观测方法。

生成测试数据(Generator)

实际运行的时候这个插件是派不上用途的，但这个插件依然是非常重要的插件之一。因为每一个使用 ELK stack 的运维人员都应该清楚一个道理：数据是支持操作的唯一真理（否则你也用不着 ELK）。所以在上线之前，你一定会需要在自己的实际环境中，测试 Logstash 和 Elasticsearch 的性能状况。这时候，这个用来生成测试数据的插件就有用了！

配置示例

```
input {
    generator {
        count => 10000000
        message => '{"key1":"value1","key2":[1,2],"key3":{"subke
y1":"subvalue1"}}'
        codec => json
    }
}
```

插件的默认生成数据，message 内容是 "hello world"。你可以根据自己的实际需要这里来写其他内容。

使用方式

做测试有两种主要方式：

- 配合 LogStash::Outputs::Null

inputs/generator 是无中生有，output/null 则是锯嘴葫芦。事件流转到这里直接就略过，什么操作都不做。相当于只测试 Logstash 的 pipe 和 filter 效率。测试过程非常简单：

```
$ time ./bin/logstash -f generator_null.conf
real    3m0.864s
user    3m39.031s
sys        0m51.621s
```

- 使用 pv 命令配合 LogStash::Outputs::Stdout 和 LogStash::Codecs::Dots

上面的这种方式虽然想法挺好，不过有个小漏洞：logstash 是在 JVM 上运行的，有一个明显的启动时间，运行也有一段事件的预热后才算稳定运行。所以，要想更真实的反应 logstash 在长期运行时候的效率，还有另一种方法：

```
output {
    stdout {
        codec => dots
    }
}
```

LogStash::Codecs::Dots 也是一个另类的 codec 插件，他的作用是：把每个 event 都变成一个点(`.`)。这样，在输出的时候，就变成了一个一个的 `.` 在屏幕上。显然这也是一个为了测试而存在的插件。

下面就要介绍 pv 命令了。这个命令的作用，就是作实时的标准输入、标准输出监控。我们这里就用它来监控标准输出：

```
$ ./bin/logstash -f generator_dots.conf | pv -abt > /dev/null
2.2MiB 0:03:00 [12.5kiB/s]
```

可以很明显的看到在前几秒中，速度是 0 B/s，因为 JVM 还没启动起来呢。开始运行的时候，速度依然不快。慢慢增长到比较稳定的状态，这时候的才是你需要的数据。

这里单位是 B/s，但是因为一个 event 就输出一个 `.` ，也就是 1B。所以 12.5kiB/s 就相当于是 12.5k event/s。

注：如果你在 *CentOS* 上通过 *yum* 安装的 *pv* 命令，版本较低，可能还不支持 *-a* 参数。单纯靠 *-bt* 参数看起来还是有点累的。

如果你要测试的是 input 插件的效率，方法也是类似的。此外，如果不想使用额外而且可能低版本的 pv 命令，通过 logstash-filter-metric 插件也可以做到类似的效果，官方博客中对此有详细阐述，建议大家阅读。

额外的话

既然单独花这么一节来说测试，这里想额外谈谈一个很常见的话题：*ELK* 的性能怎么样？

其实这压根就是一个不正确的提问。ELK 并不是一个软件而是一个并不耦合的套件。所以，我们需要分拆开讨论这三个软件的性能如何？怎么优化？

- LogStash 的性能，是最让新人迷惑的地方。因为 LogStash 本身并不维护队列，所以整个日志流转中任意环节的问题，都可能看起来像是 LogStash 的问题。这里，需要熟练使用本节说的测试方法，针对自己的每一段配置，都确定其性能。另一方面，就是本书之前提到过的，LogStash 给自己的线程都设置了单独的线程名称，你可以在 `top -H` 结果中查看具体线程的负载情况。
- Elasticsearch 的性能。这里最需要强调的是：Elasticsearch 是一个分布式系统。从来没有分布式系统要跟人比较单机处理能力的说法。所以，更需要关注的是：在确定的单机处理能力的前提下，性能是否能做到线性扩展。当然，这不意味着说提高处理能力只能靠加机器了——有效利用 mapping API 是非常重要的。不过暂时就在这里讲述了。
- Kibana 的性能。通常来说，Kibana 只是一个单页 Web 应用，只需要 nginx 发布静态文件即可，没什么性能问题。页面加载缓慢，基本上是因为 Elasticsearch 的请求响应时间本身不够快导致的。不过一定要细究的话，也能找出点 Kibana 本身性能相关的话题：因为 Kibana3 默认是连接固定的一个 ES 节点的 IP 端口的，所以这里会涉及一个浏览器的同一 IP 并发连接数的限制。其次，就是 Kibana 用的 AngularJS 使用了 Promise.then 的方式来处理 HTTP 请求响应。这是异步的。

心跳检测

缺少内部队列状态的监控办法一直是 logstash 最为人诟病的一点。从 logstash-1.5.1 版开始，新发布了一个 logstash-input-heartbeat 插件，实现了一个最基本的队列堵塞状态监控。

配置示例如下：

```
input {
    heartbeat {
        message => "epoch"
        interval => 60
        type => "heartbeat"
        add_field => {
            "zbxkey" => "logstash.heartbeat",
            "zbxhost" => "logstash_hostname"
        }
    }
    tcp {
        port => 5160
    }
}
output {
    if [type] == "heartbeat" {
        file {
            path => "/data1/logstash-log/local6-5160-%{+YYYY.MM.
dd}.log"
        }
        zabbix {
            zabbix_host => "zbxhost"
            zabbix_key => "zbxkey"
            zabbix_server_host => "zabbix.example.com"
            zabbix_value => "clock"
        }
    } else {
        elasticsearch { }
    }
}
```

示例中，同时将数据输出到本地文件和 zabbix server。注意，logstash-output-zabbix 并不是标准插件，需要额外安装：

```
bin/logstash-plugin install logstash-output-zabbix
```

文件中记录的就是 heartbeat 事件的内容，示例如下：

```
{"clock":1435191129,"host":"logtes004.mweibo.bx.sinanode.com","@version":"1","@timestamp":"2015-06-25T00:12:09.042Z","type":"heartbeat","zbxkey":"logstash.heartbeat","zbxhost":"logstash_hostname"}
```


可以通过文件最后的 clock 和 @timestamp 内容，对比当前时间，来判断 logstash 内部队列是否有堵塞。

JMX 监控方式

Logstash 是一个运行在 JVM 上的软件，也就意味着 JMX 这种对 JVM 的通用监控方式对 Logstash 也是一样有效果的。要给 Logstash 启用 JMX，需要修改 `./bin/logstash.lib.sh` 中 `$JAVA_OPTS` 变量的定义，或者在运行时设置 `LS_JAVA_OPTS` 环境变量。

在 `./bin/logstash.lib.sh` 第 34 行 `JAVA_OPTS="$JAVA_OPTS -Djava.awt.headless=true"` 下，添加如下几行：

```
    JAVA_OPTS="$JAVA_OPTS -Dcom.sun.management.jmxremote"
    JAVA_OPTS="$JAVA_OPTS -Dcom.sun.management.jmxremote.port=90
10"
    JAVA_OPTS="$JAVA_OPTS -Dcom.sun.management.jmxremote.local.o
nly=false"
    JAVA_OPTS="$JAVA_OPTS -Dcom.sun.management.jmxremote.authent
icate=false"
    JAVA_OPTS="$JAVA_OPTS -Dcom.sun.management.jmxremote.ssl=fal
se"
```

重启 logstash 服务，JMX 配置即可生效。

有 JMX 以后，我们可以通过 jconsole 界面查看，也可以通过 zabbix 等监控系统做长期监控。甚至 logstash 自己也有插件 logstash-input-jmx 来读取远程 JMX 数据。

zabbix 监控

zabbix 里提供了专门针对 JMX 的监控项。详见:https://www.zabbix.com/documentation/2.2/manual/config/items/itemtypes/jmx_monitoring

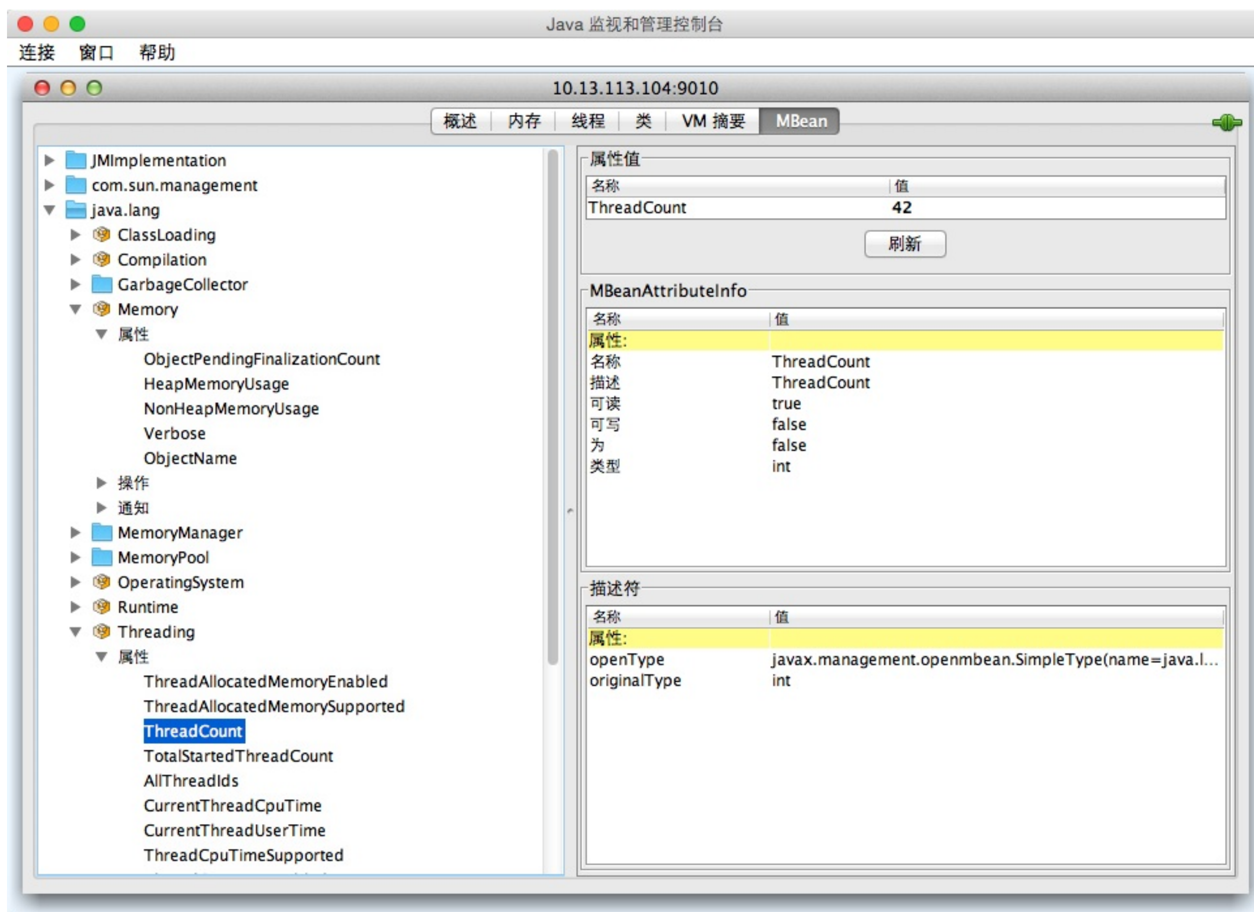
注意，zabbix-server 本身并不直接对 JMX 发起请求，而是单独有一个 Java Gateway 作为中间代理层角色。zabbix-server 的 java poller 连接 zabbix-java-gateway，由 zabbix-java-gateway 去获取远程 JMX 信息。所以，在 zabbix-web 配置之前，需要先配置 zabbix server 相关进程和设置：

```
# yum install zabbix-java-gateway
# cat >> /etc/zabbix/zabbix-server.conf <<EOF
JavaGateway=127.0.0.1
JavaGatewayPort=10052
StartJavaPollers=5
EOF
# /etc/init.d/zabbix-java-gateway restart
# /etc/init.d/zabbix-server restart
```

然后在 zabbix-web 上 **Configuration** 页，给运行 logstash 的主机的 **Host** 配置添加 **JMX interfaces**，**Port** 即为上面定义的 9010 端口。

最后添加 Item，**Type** 下拉框选择 `JMX agent`，**Key** 文本框输入 `jmx["java.lang:type=Memory", "HeapMemoryUsage.used"]`，保存即可。

JMX 有很多 Key 可以监控，具体的值，可以通过 jconsole 参看。如下图所示，如果要监控线程数，就可以写成 `jmx["java.lang:type=Threading", "ThreadCount"]`。



有了监控项和数据，后续的 Graph, Screen, Trigger 定义，这里就不再讲述了，有需要的读者可以自行查找 Zabbix 相关资料。

Logstash 的监控 API

Logstash 5.0 开始，提供了输出自身进程的指标和状态监控的 API。这大大降低了我们监控 Logstash 的难度。

目前 API 主要有四类：

- 节点信息
- 插件信息
- 节点指标
- 热线程统计

节点信息

node info 接口目前支持三种类型：pipeline、os、jvm。没什么要紧的。

插件信息

用来列出已安装插件的名称和版本。

节点指标

node stats 接口目前支持四种类型的指标：

events

获取该指标的方式为：

```
curl -s localhost:9600/_node/stats/events?pretty=true
```

是的，Logstash 跟 Elasticsearch 一样也支持用 `?pretty` 参数美化 JSON 输出。此外，还支持 `?format=yaml` 来输出 YAML 格式的指标统计。Logstash 默认监听在 9600 端口提供这些 API 访问。如果需要修改，通过 `--http.port` 命令行参数，或者对应的 `logstash.yml` 设置修改。

该指标的响应结果示例如下：

```
{
    "events" : {
        "in" : 59685,
        "filtered" : 59685,
        "out" : 59685
    }
}
```

jvm

获取该指标的方式为：

```
curl -s localhost:9600/_node/stats/jvm?pretty=true
```

该指标的响应结果示例如下：

```
{
    "jvm" : {
        "threads" : {
            "count" : 32,
            "peak_count" : 34
        }
    }
}
```

process

获取该指标的方式为：

```
curl -s localhost:9600/_node/stats/process?pretty=true
```

该指标的响应结果示例如下：

```
{
    "process" : {
            "peak_open_file_descriptors" : 64,
            "max_file_descriptors" : 10240,
            "open_file_descriptors" : 64,
        "mem" : {
            "total_virtual_in_bytes" : 5278068736
        },
        "cpu" : {
            "total_in_millis" : 103290097000,
            "percent" : 0
        }
    }
}
```

目前 beats 家族有个 logstashbeat 项目，就是专门采集这个数据的。

pipeline

获取该指标的方式为：

```
curl -s localhost:9600/_node/stats/pipeline?pretty=true
```

该指标的响应结果示例如下：

```
{
    "pipeline": {
        "events": {
            "duration_in_millis": 7863504,
            "in": 100,
            "filtered": 100,
            "out": 100
        },
        "plugins": {
            "inputs": [],
            "filters": [
                {
                    "id": "grok_20e5cb7f7c9e712ef9750edf94aefb46
```

```
5e3e361b-2",
                    "events": {
                        "duration_in_millis": 48,
                        "in": 100,
                        "out": 100
                    },
                    "matches": 100,
                    "patterns_per_field": {
                        "message": 1
                    },
                    "name": "grok"
                },
                {
                    "id": "geoip_20e5cb7f7c9e712ef9750edf94aefb4
65e3e361b-3",
                    "events": {
                        "duration_in_millis": 141,
                        "in": 100,
                        "out": 100
                    },
                    "name": "geoip"
                }
            ],
            "outputs": [
                {
                    "id": "20e5cb7f7c9e712ef9750edf94aefb465e3e3
61b-4",
                    "events": {
                        "in": 100,
                        "out": 100
                    },
                    "name": "elasticsearch"
                }
            ]
        },
        "reloads": {
            "last_error": null,
            "successes": 0,
            "last_success_timestamp": null,
            "last_failure_timestamp": null,
```

```
                "failures": 0
            }
        }
}
```

可以看到它这里显示了每个插件的日志处理情况(数量、耗时等)，尤其是 grok 过滤器插件，还显示出来了正则匹配失败的数量、每个字段匹配的正则表达式个数等很有用的排障和性能调优信息。

热线程统计

上面的指标值可能比较适合的是长期趋势的监控，在排障的时候，更需要的是即时的线程情况统计。获取方式如下：

```
curl -s localhost:9600/_node/stats/hot_threads?human=true
```

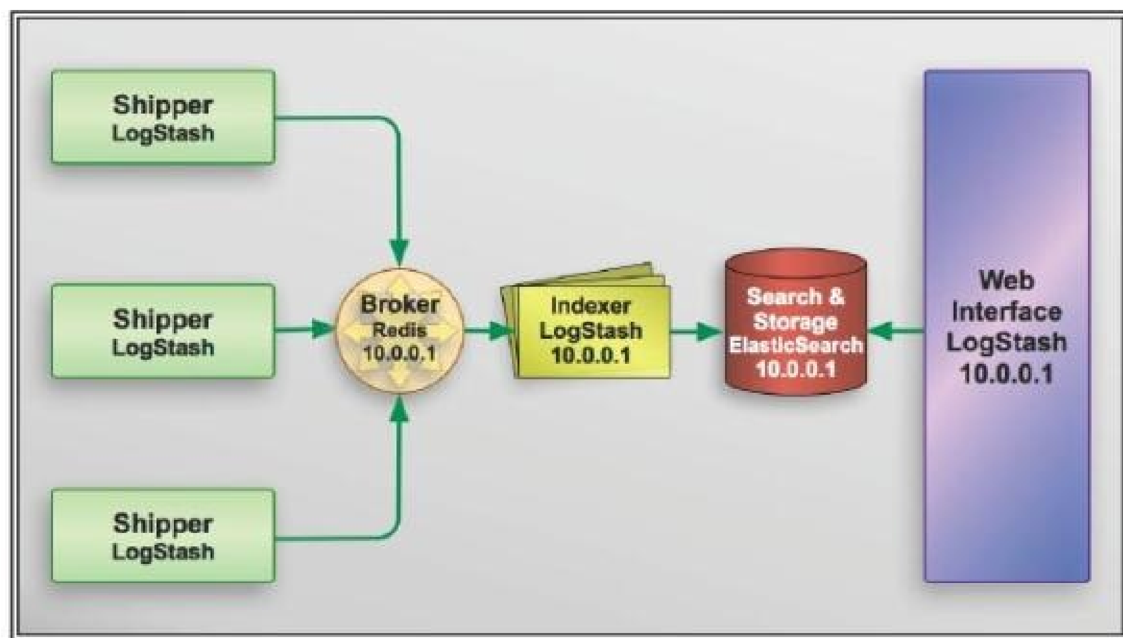
该接口默认返回也是 JSON 格式，在看堆栈的时候并不方便，可以用 `?human=true` 参数来改成文本换行的样式。效果上跟我们看 Elasticsearch 的 `/_nodes/_local/hot_threads` 效果就一样了。

其实节点指标 API 也有 `?human=true` 参数，其作用和 `hot_threads` 不一样，是把一些网络字节数啊，时间啊，改成人类更易懂的大单位。

扩展方案

之前章节中，讲述的都是单个 logstash 进程，如何配置实现对数据的读取、解析和输出处理。但是在生产环境中，从每台应用服务器运行 logstash 进程并将数据直接发送到 Elasticsearch 里，显然不是第一选择：第一，过多的客户端连接对 Elasticsearch 是一种额外的压力；第二，网络抖动会影响到 logstash 进程，进而影响生产应用；第三，运维人员未必愿意在生产服务器上部署 Java，或者让 logstash 跟业务代码争夺 Java 资源。

所以，在实际运用中，logstash 进程会被分为两个不同的角色。运行在应用服务器上的，尽量减轻运行压力，只做读取和转发，这个角色叫做 shipper；运行在独立服务器上，完成数据解析处理，负责写入 Elasticsearch 的角色，叫 indexer。



logstash 作为无状态的软件，配合消息队列系统，可以很轻松的做到线性扩展。本节首先会介绍最常见的两个消息队列与 logstash 的配合。

此外，logstash 作为一个框架式的项目，并不排斥，甚至欢迎与其他类似软件进行混搭式的运行。本节也会介绍一些其他日志处理框架以及如何和 logstash 共存的方式（《logstashbook》也同样有类似内容）。希望大家各取所长，做好最适合自己的日志处理系统。

利用 Redis 队列扩展 logstash

Redis 服务器是 logstash 官方推荐的 broker 选择。Broker 角色也就意味着会同时存在输入和输出俩个插件。

读取 Redis 数据

`LogStash::Inputs::Redis` 支持三种 *data_type*（实际上是*redis_type*），不同的数据类型会导致实际采用不同的 Redis 命令操作：

- list => BLPOP
- channel => SUBSCRIBE
- pattern_channel => PSUBSCRIBE

注意到了么？这里面没有 **GET** 命令！

Redis 服务器通常都是用作 NoSQL 数据库，不过 logstash 只是用来做消息队列。所以不要担心 logstash 里的 Redis 会撑爆你的内存和磁盘。

配置示例

```
input {
    redis {
        data_type => "pattern_channel"
        key => "logstash-*"
        host => "192.168.0.2"
        port => 6379
        threads => 5
    }
}
```

使用方式

基本方法

首先确认你设置的 host 服务器上已经运行了 redis-server 服务，然后打开终端运行 logstash 进程等待输入数据，然后打开另一个终端，输入 `redis-cli` 命令(先安装好 redis 软件包)，在交互式提示符后面输入 `PUBLISH logstash-demochan "hello world"` ：

```
# redis-cli
127.0.0.1:6379> PUBLISH logstash-demochan "hello world"
```

你会在第一个终端里看到 logstash 进程输出类似下面这样的内容：

```
{
       "message" => "hello world",
      "@version" => "1",
    "@timestamp" => "2014-08-08T16:26:29.399Z"
}
```

注意：这个事件里没有 **host** 字段！（或许这算是 bug……）

输入 JSON 数据

如果你想通过 redis 的频道给 logstash 事件添加更多字段，直接向频道发布 JSON 字符串就可以了。 `LogStash::Inputs::Redis` 会直接把 JSON 转换成事件。

继续在第二个终端的交互式提示符下输入如下内容：

```
127.0.0.1:6379> PUBLISH logstash-chan '{"message":"hello world",
"@version":"1","@timestamp":"2014-08-08T16:34:21.865Z","host":"r
aochenlindeMacBook-Air.local","key1":"value1"}'
```

你会看到第一个终端里的 logstash 进程随即也返回新的内容，如下所示：

```
{
       "message" => "hello world",
      "@version" => "1",
    "@timestamp" => "2014-08-09T00:34:21.865+08:00",
          "host" => "raochenlindeMacBook-Air.local",
          "key1" => "value1"
}
```

看，新的字段出现了！现在，你可以要求开发工程师直接向你的 redis 频道发送信息好了，一切自动搞定。

小贴士

这里我们建议的是使用 *pattern_channel* 作为输入插件的 *data_type* 设置值。因为实际使用中，你的 redis 频道可能有很多不同的 *keys*，一般命名成 *logstash-chan-%{type}* 这样的形式。这时候 *pattern_channel* 类型就可以帮助你一次订阅全部 logstash 相关频道！

扩展方式

如上段"小贴士"提到的，之前两个使用场景采用了同样的配置，即数据类型为频道发布订阅方式。这种方式在需要扩展 logstash 成多节点集群的时候，会出现一个问题：通过频道发布的一条信息，会被所有订阅了该频道的 **logstash** 进程同时接收到，然后输出重复内容！

你可以尝试再做一次上面的实验，这次在两个终端同时启动 `logstash -f redis-input.conf` 进程，结果会是两个终端都输出消息。

这种时候，就需要用 *list* 类型。在这种类型下，数据输入到 redis 服务器上暂存，logstash 则连上 redis 服务器取走 (`BLPOP` 命令，所以只要 logstash 不堵塞，redis 服务器上也不会有数据堆积占用空间)数据。

配置示例

```
input {
    redis {
        batch_count => 1
        data_type => "list"
        key => "logstash-list"
        host => "192.168.0.2"
        port => 6379
        threads => 5
    }
}
```

使用方式

这次我们同时在两个终端运行 `logstash -f redis-input-list.conf` 进程。然后在第三个终端里启动 redis-cli 命令交互：

```
$ redis-cli
127.0.0.1:6379> RPUSH logstash-list "hello world"
(integer) 1
```

这时候你可以看到，只有一个终端输出了结果。

连续 `RPUSH` 几次，可以看到两个终端近乎各自输出一半条目。

小贴士

RPUSH 支持 batch 方式，修改 logstash 配置中的 `batch_count` 值，作为示例这里只改到 2，实际运用中可以更大(事实上 `LogStash::Outputs::Redis` 对应这点的 `batch_event` 配置默认值就是 50)。

重启 logstash 进程后，redis-cli 命令中改成如下发送：

```
127.0.0.1:6379> RPUSH logstash-list "hello world" "hello world"
"hello world" "hello world" "hello world" "hello world"
(integer) 3
```

可以看到，两个终端也各自输出一部分结果。而你只用了一次 RPUSH 命令。

输出到 Redis

配置示例

```
input { stdin {} }
output {
    redis {
        data_type => "channel"
        key => "logstash-chan-%{+yyyy.MM.dd}"
    }
}
```

使用方式

我们还是继续先用 `redis-cli` 命令行来演示 *outputs/redis* 插件的实质。

基础方式

运行 logstash 进程，然后另一个终端启动 redis-cli 命令。输入订阅指定频道的 Redis 命令 ("SUBSCRIBE logstash-chan-2014.08.08") 后，首先会看到一个订阅成功的返回信息。如下所示：

```
# redis-cli
127.0.0.1:6379> SUBSCRIBE logstash-chan-2014.08.08
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "logstash-chan-2014.08.08"
3) (integer) 1
```

好，在运行 logstash 的终端里输入 "hello world" 字符串。切换回 redis-cli 的终端，你发现已经自动输出了一条信息：

```
1) "message"
2) "logstash-chan-2014.08.08"
3) "{\"message\":\"hello world\",\"@version\":\"1\",\"@timestamp
\":\"2014-08-08T16:34:21.865Z\",\"host\":\"raochenlindeMacBook-A
ir.local\"}"
```

broker 方式

上面那条信息看起来是不是非常眼熟？这一串字符其实就是我们在前面"读取 Redis 中的数据"小节中使用的那段数据。

看，这样就把 *outputs/redis* 和 *inputs/redis* 串联起来了吧！

事实上，这就是我们使用 redis 服务器作为 logstassh 架构中 broker 角色的原理。

让我们把这两节中不同配置的 logstash 进程分别在两个终端运行起来，这次不再要运行 redis-cli 命令了。在配有 *outputs/redis* 这端输入 "hello world"，配有 "inputs/redis" 的终端上，就自动输出数据了！

告警用途

我们还可以用其他程序来订阅 redis 频道，程序里就可以随意写其他逻辑了。你可以看看 output/juggernaut 插件的原理。这个 Juggernaut 就是基于 redis 服务器和 socket.io 框架构建的。利用它，logstash 可以直接向 webkit 等支持 socket.io 的浏览器推送告警信息。

扩展方式

和 `LogStash::Inputs::Redis` 一样，这里也有设置成 **list** 的方式。使用 `RPUSH` 命令发送给 redis 服务器，效果和之前展示的完全一致。包括可以调整的参数 `batch_event` ，也在之前章节中讲过。这里不再重复举例。

通过kafka传输

本节作者：*jingbli*

Kafka 是一个高吞吐量的分布式发布订阅日志服务，具有高可用、高性能、分布式、高扩展、持久性等特性。目前已经在各大公司中广泛使用。和之前采用 Redis 做轻量级消息队列不同，Kafka 利用磁盘作队列，所以也就无所谓消息缓冲时的磁盘问题。此外，如果公司内部已有 Kafka 服务在运行，logstash 也可以快速接入，免去重复建设的麻烦。

如果打算新建 Kafka 系统的，请参考 Kafka 官方入门文档：http://kafka.apache.org/documentation.html#quickstart

kafka 基本概念

以下仅对相关基本概念说明，更多概念见官方文档：

- Topic 主题，声明一个主题，producer指定该主题发布消息，订阅该主题的consumer对该主题进行消费
- Partition 每个主题可以分为多个分区，每个分区对应磁盘上一个目录，分区可以分布在不同broker上，producer在发布消息时，可以通过指定partition key映射到对应分区，然后向该分区发布消息，在无partition key情况下，随机选取分区，一段时间内触发一次(比如10分钟)，这样就保证了同一个producer向同一partition发布的消息是顺序的。 消费者消费时，可以指定partition进行消费，也可以使用high-level-consumer api,自动进行负载均衡，并将partition分给consumer，一个partition只能被一个consumer进行消费。
- Consumer 消费者，可以多实例部署，可以批量拉取，有两类API可供选择：一个simpleConsumer，暴露所有的操作给用户，可以提交offset、fetch offset、指定partition fetch message；另外一个high-level-consumer(ZookeeperConsumerConnector)，帮助用户做基于partition自动分配的负载均衡，定期提交offset，建立消费队列等。simpleConsumer相当于手动挡，high-level-consumer相当于自动挡。

  simpleConsumer：无需像high-level-consumer那样向zk注册brokerid、owner，甚至不需要提交offset到zk，可以将offset提交到任意地方比如(mysql,本地文件等)。

high-level-consumer：一个进程中可以启多个消费线程，一个消费线程即是一个consumer，假设A进程里有2个线程(consumerid分别为1，2)，B进程有2个线程(consumerid分别为1，2)，topic1的partition有5个，那么partition分配是这样的：

```
partition1 ---> A进程consumerid1
partition2 ---> A进程consumerid1
partition3 ---> A进程consumerid2
partition4 ---> B进程consumer1
partition5 ---> B进程consumer2
```

- Group High-level-consumer可以声明group，每个group可以有多个consumer，每group各自管理各自的消费offset，各个不同group之间互不关联影响。

  由于目前版本消费的offset、owner、group都是consumer自己通过zk管理，所以group对于broker和producer并不关心，一些监控工具需要通过group来监控，simpleComsumer无需声明group。

小提示

以上概念是 logstash 的 kafka 插件的必要参数，请理解阅读，对后续使用 kafka 插件有重要作用。logstash-kafka-input 插件使用的是 `High-level-consumer API` 。

插件安装

logstash-1.4 安装

如果你使用的还是 1.4 版本，需要自己单独安装 logstash-kafka 插件。插件地址见：https://github.com/joekiller/logstash-kafka。

插件本身内容非常简单，其主要依赖同一作者写的 jruby-kafka 模块。需要注意的是：该模块仅支持 **Kafka-0.8** 版本。如果是使用 **0.7** 版本 **kafka** 的，将无法直接使 **jruby-kafka** 该模块和 **logstash-kafka** 插件。

安装按照官方文档完全自动化的安装。或是可以通过以下方式手动自己安装插件，不过重点注意的是 **kafka** 的版本，上面已经指出了。

1. 下载 logstash 并解压重命名为 `./logstash-1.4.0` 文件目录。
2. 下载 kafka 相关组件，以下示例选的为 kafka_2.8.0-0.8.1.1-src，并解压重命名为 `./kafka_2.8.0-0.8.1.1` 。
3. 从 releases 页下载 logstash-kafka v0.4.2 版，并解压重命名为 `./logstash-kafka-0.4.2` 。
4. 从 `./kafka_2.8.0-0.8.1.1/libs` 目录下复制所有的 jar 文件拷贝到 `./logstash-1.4.0/vendor/jar/kafka_2.8.0-0.8.1.1/libs` 下，其中你需要创建 `kafka_2.8.0-0.8.1.1/libs` 相关文件夹及目录。
5. 分别复制 `./logstash-kafka-0.4.2/logstash` 里的 `inputs` 和 `outputs` 下的 `kafka.rb` ，拷贝到对应的 `./logstash-1.4.0/lib/logstash` 里的 `inputs` 和 `outputs` 对应目录下。
6. 切换到 `./logstash-1.4.0` 目录下，现在需要运行 logstash-kafka 的 gembag.rb 脚本去安装 jruby-kafka 库，执行以下命令： `GEM_HOME=vendor/bundle/jruby/1.9 GEM_PATH= java -jar vendor/jar/jruby-complete-1.7.11.jar --1.9 ../logstash-kafka-0.4.2/gembag.rb ../logstash-kafka-0.4.2/logstash-kafka.gemspec` 。
7. 现在可以使用 logstash-kafka 插件运行 logstash 了。

logstash-1.5 安装

logstash 从 1.5 版本开始才集成了 Kafka 支持。1.5 版本开始所有插件的目录和命名都发生了改变，插件发布地址见：https://github.com/logstash-plugins。 安装和更新插件都可以使用官方提供的方式：

```
$bin/plugin install OR $bin/plugin update
```

小贴士

对于插件的安装和更新，默认走的Gem源为 https://rubygems.org,对于咱们国内网络来说是出奇的慢或是根本无法访问（爬梯子除外），在安装或是更新插件是，可以尝试修改目录下 `Gemfile` 文件中的 `source` 为淘宝源 https://ruby.taobao.org，这样会使你的安装或是更新顺畅很多。

插件配置

Input 配置示例

以下配置可以实现对 kafka 读取端(consumer)的基本使用。

消费端更多详细的配置请查看 http://kafka.apache.org/documentation.html#consumerconfigs kafka 官方文档的消费者部分配置文档。

```
input {
    kafka {
        zk_connect => "localhost:2181"
        group_id => "logstash"
        topic_id => "test"
        codec => plain
        reset_beginning => false # boolean (optional)， default:
 false
        consumer_threads => 5  # number (optional)， default: 1
        decorate_events => true # boolean (optional)， default:
false
    }
}
```

Input 解释

作为 Consumer 端,插件使用的是 `High-level-consumer API` ，请结合上述 **kafka** 基本概念进行设置：

- group_id

消费者分组，可以通过组 ID 去指定，不同的组之间消费是相互不受影响的，相互隔离。

- topic_id

指定消费话题，也是必填项目，指定消费某个 `topic` ，这个其实就是订阅某个主题，然后去消费。

- reset_beginning

logstash 启动后从什么位置开始读取数据，默认是结束位置，也就是说 logstash 进程会以从上次读取结束时的偏移量开始继续读取，如果之前没有消费过，那么就开始从头读取.如果你是要导入原有数据，把这个设定改成 "true"， logstash 进程就从头开始读取.有点类似 `cat` ，但是读到最后一行不会终止，而是变成 `tail -F` ，继续监听相应数据。

- decorate_events

在输出消息的时候会输出自身的信息包括:消费消息的大小， topic 来源以及 consumer 的 group 信息。

- rebalance_max_retries

当有新的 consumer(logstash) 加入到同一 group 时，将会 `reblance` ，此后将会有 `partitions` 的消费端迁移到新的 `consumer` 上，如果一个 `consumer` 获得了某个 `partition` 的消费权限，那么它将会向 `zookeeper` 注册， `Partition Owner registry` 节点信息，但是有可能此时旧的 `consumer` 尚没有释放此节点，此值用于控制，注册节点的重试次数。

- consumer_timeout_ms

指定时间内没有消息到达就抛出异常，一般不需要改。

以上是相对重要参数的使用示例，更多参数可以选项可以跟据 https://github.com/joekiller/logstash-kafka/blob/master/README.md 查看 input 默认参数。

注意

1.想要使用多个 logstash 端协同消费同一个 `topic` 的话，那么需要把两个或是多个 logstash 消费端配置成相同的 `group_id` 和 `topic_id` ，但是前提是要把相应的 **topic** 分多个 **partitions (**区**)**，多个消费者消费是无法保证消息的消费顺序性的。

> 这里解释下，为什么要分多个 **partitions(**区**)**， kafka 的消息模型是对 topic 分区以达到分布式效果。每个 `topic` 下的不同的 **partitions (**区**)**只能有一个 **Owner** 去消费。所以只有多个分区后才能启动多个消费者，对应不同的区去消费。其中协调消费部分是由 server 端协调而成。不必使用者考虑太多。只是消息的消费则是无序的。

总结:保证消息的顺序，那就用一个 **partition**。 **kafka** 的每个 **partition** 只能同时被同一个 **group** 中的一个 **consumer** 消费。

Output 配置

以下配置可以实现对 kafka 写入端 (producer) 的基本使用。

生产端更多详细的配置请查看 http://kafka.apache.org/documentation.html#producerconfigs kafka 官方文档的生产者部分配置文档。

```
 output {
    kafka {
        bootstrap_servers => "localhost:9092"
        topic_id => "test"
        compression_codec => "snappy" # string (optional)， one
of ["none"， "gzip"， "snappy"]， default: "none"
    }
}
```

Output 解释

作为 Producer 端使用，以下仅为重要概念解释，请结合上述 **kafka** 基本概念进行设置：

- compression_codec

消息的压缩模式，默认是 none，可以有 gzip 和 snappy (暂时还未测试开启压缩与不开启的性能，数据传输大小等对比)。

- compressed_topics

可以针对特定的 topic 进行压缩，设置这个参数为 `topic` ，表示此 `topic` 进行压缩。

- request_required_acks

消息的确认模式:

> 可以设置为 0: 生产者不等待 broker 的回应，只管发送.会有最低能的延迟和最差的保证性(在服务器失败后会导致信息丢失) 可以设置为 1: 生产者会收到 leader 的回应在 leader 写入之后.(在当前 leader 服务器为复制前失败可能会导致信息丢失) 可以设置为 -1: 生产者会收到 leader 的回应在全部拷贝完成之后。

- partitioner_class

分区的策略，默认是 hash 取模

- send_buffer_bytes

socket 的缓存大小设置，其实就是缓冲区的大小

消息模式相关

- serializer_class

消息体的系列化处理类，转化为字节流进行传输，请注意 **encoder** 必须和下面的 **`key_serializer_class`** 使用相同的类型。

- key_serializer_class

默认的是与 `serializer_class` 相同

- producer_type

生产者的类型 `async` 异步执行消息的发送 `sync` 同步执行消息的发送

- queue_buffering_max_ms

异步模式下，那么就会在设置的时间缓存消息，并一次性发送

- queue_buffering_max_messages

异步的模式下，最长等待的消息数

- queue_enqueue_timeout_ms

异步模式下，进入队列的等待时间，若是设置为0，那么要么进入队列，要么直接抛弃

- batch_num_messages

异步模式下，每次发送的最大消息数，前提是触发了 `queue_buffering_max_messages` 或是 `queue_enqueue_timeout_ms` 的限制

以上是相对重要参数的使用示例，更多参数可以选项可以跟据 https://github.com/joekiller/logstash-kafka/blob/master/README.md 查看 output 默认参数。

小贴士

logstash-kafka 插件输入和输出默认 `codec` 为 json 格式。在输入和输出的时候注意下编码格式。消息传递过程中 logstash 默认会为消息编码内加入相应的时间戳和 hostname 等信息。如果不想要以上信息(一般做消息转发的情况下)，可以使用以下配置，例如:

```
 output {
    kafka {
        codec => plain {
            format => "%{message}"
        }
    }
}
```

作为 Consumer 从kafka中读数据，如果为非 json 格式的话需要进行相关解码，例如：

```
input {
    kafka {
        zk_connect => "xxx：xxx"
        group_id => "test"
        topic_id => "test-topic"
        codec => "line"
        ...........
    }
}
```

性能

队列监控

其实 logstash 的 kafka 插件性能并不是很突出，可以通过使用以下命令查看队列积压消费情况：

```
$/bin/kafka-run-class.sh kafka.tools.ConsumerOffsetChecker --gro
up test
```

队列积压严重，性能跟不上的情况下，结合实际服务器资源，可以适当增加 topic 的 partition 多实例化 Consumer 进行消费处理消息。

input-kafka 的 JSON 序列化性能

此外，跟 logstash-input-syslog 改在 filter 阶段 grok 的优化手段类似，也可以将 logstash-input-kafka 的默认 JSON 序列化操作从 codec 阶段后移到 filter 阶段。如下：

```
input {
    kafka {
        codec => plain
    }
}
filter {
    json {
        source => "message"
    }
}
```

然后通过 `bin/logstash -w $num_cpus` 运行，利用多核计算优势，可以获得大约一倍左右的性能提升。

其他方案

- https://github.com/reachkrishnaraj/kafka-elasticsearch-standalone-consumer
- https://github.com/childe/hangout

AIX 平台上的 Logstash-Forwarder-Java

本节作者：鹏程

在AIX环境下（IBM Power小型机的一种操作系统）,你无法使用logstash（因为IBM JDK没有实现相关方法），也无法使用logstash-forwarder，github上有个Logstash-forwarder再实现的项目就是解决这个问题的：https://github.com/didfet/logstash-forwarder-java

配置和Logstash-forwarder基本一致，但是注意有一个坑是需要关注的，作者也在他的github上提到了，就是：

```
the ssl ca parameter points to a java keystore containing the root certificate of the server, not a PEM file
```

不熟悉证书相关体系的读者可能不太清楚这个意思，换句话说，如果你还按照logstash-forwarder的配置方法配置shipper端，那么你将会得到一个诡异的java.io.IOException: Invalid keystore format 异常。

首先介绍下背景知识，摘录一段知乎上的讲解：@刘长元 from http://www.zhihu.com/question/29620953

> 把SSL系统比喻为工商局系统。 首先有SSL就有CA，certificate authority。证书局，用于制作、认证证书的第三方机构，我们假设营业执照非常难制作，就像身份证一样，需要有制证公司来提供，并且提供技术帮助工商局验证执照的真伪。然后CA是可以有多个的，也就是可以有多个制证公司，但工商局就只有一个，它来说那个制证公司是可信的，那些是假的，需要打击。在SSL的世界中，微软、Google和Mozilla扮演了一部分这个角色。也就是说，IE、Chrome、Firefox中内置有一些CA，经过这些CA颁发，验证过的证书都是可以信的，否则就会提示你不安全。 这也是为什么前几天Chrome决定屏蔽CNNIC的CA时，CNNIC那么遗憾了。 也因为内置的CA是相对固定的，所以当你决定要新建网站时，就需要购买这些内置CA颁发的证书来让用户看到你的域名前面是绿色的，而不是红色。而这个最大的卖证书的公司就是VeriSign如果你听说过的话，当然它被卖给了Symantec，这家伙不只出Ghost，还是个卖证书的公司。

> 要开店的老板去申请营业执照的时候是需要交他的身份证的，然后办出来的营业执照上也会有他的照片和名字。身份证相当于私钥，营业执照就是证书，Ceritficate，.cer文件。
>
> 然后关于私钥和公钥如何解释我没想好，而它们在数据加密层面，数据的流向是这样的。
>
> 消息-->[公钥]-->加密后的信息-->[私钥]-->消息
>
> 公钥是可以随便扔给谁的，他把消息加了密传给我。对了，可以这样理解，我有一个箱子，一把锁和一把钥匙，我把箱子和开着的锁给别人，他写了信放箱子里，锁上，然后传递回我手上的途中谁都是打不开箱子的，只有我可以用原来的钥匙打开，这就是SSL，公钥，私钥传递加密消息的方式。这里的密钥就是key文件。
>
> 于是我们就有了.cer和.key文件。接下来说keystore
>
> 不同的语言、工具序列SSL相关文件的格式和扩展名是不一样的。 其中Java系喜欢用keystore, truststore来干活，你看它的名字，Store，仓库，它里面存放着key和信任的CA，key和CA可以有多个。 这里的truststore就像你自己电脑的证书管理器一样，如果你打开Chrome的设置，找到HTTP SSL，就可以看到里面有很多CA，truststore就是干这个活儿的，它也里面也是存一个或多个CA让Tomcat或Java程序来调用。 而keystore就是用来存密钥文件的，可以存放多个。
>
> 然后是PEM，它是由RFC1421至1424定义的一种数据格式。其实前面的.cert和.key文件都是PEM格式的，只不过在有些系统中（比如Windows）会根据扩展名不同而做不同的事。所以当你看到.pem文件时，它里面的内容可能是certificate也可能是key，也可能两个都有，要看具体情况。可以通过openssl查看。

看到这儿你就应该懂了，按照logstash-forwarder-java的作者设计，此时你的shipper端ssl ca这个域配置的应该是keystore，而不是PEM，因此需要从你生成的crt中创建出keystore（jks）文件，方法为：

```
keytool -importcert -trustcacerts -file logstash-forwarder.crt -
alias ca -keystore keystore.jks
```

一个示例的shipper.conf为：

```
{
  "network": {
    "servers": [ "192.168.1.1:5043"],
    "ssl ca": "/mnt/disk12/logger/logstash/config/keystore.jks"
  },
  "files": [
    {
      "paths": [ "/mnt/disk12/logger/logstash/config/2.txt" ],
      "fields": { "type": "sadb" }
    }
  ]
}
```

注意：*server*可以配置多个，这样*agent*如果一个*logstash*连接不上可以连接另外的。

其余配置信息，请参考logstash-forwarder，它完全兼容。主要包括下面几个可用配置项：

- network.servers: 用来指定远端(即 logstash indexer 角色)服务器的 IP 地址和端口。这里可以写数组，但是 logstash-forwarder 只会随机选一台作为对端发送数据，一直到对端故障，才会重选其他服务器。
- network.ssl*: 网络交互中使用的 SSL 证书路径。
- files.*.paths: 读取的文件路径。 logstash-forwarder 只支持两种输入，一种就是示例中用的文件方式，和 logstash 一样也支持 glob 路径，即 `"/var/log/*.log"` 这样的写法；一种是标准输入，写法为 `"paths": [ "-" ]`
- files.*.fields: 给每条日志添加的固定字段，相当于 logstash 里的 `add_field` 参数。

配置好以后启动它即可： `nohup java -jar logstash-forwarder-java-0.2.3-SNAPSHOT.jar -quiet -config logforwarder.conf > logforwarder.log &`

-quiet 参数可以大大减少不必要的日志量，如果遇到错误请打开*-debug*和*-trace*选项，得到相关信息后查证，未果时请转向该项目*github*的*issue*区，作者很热心

测试通过环境：

- AIX版本

> 6100-04-06-1034

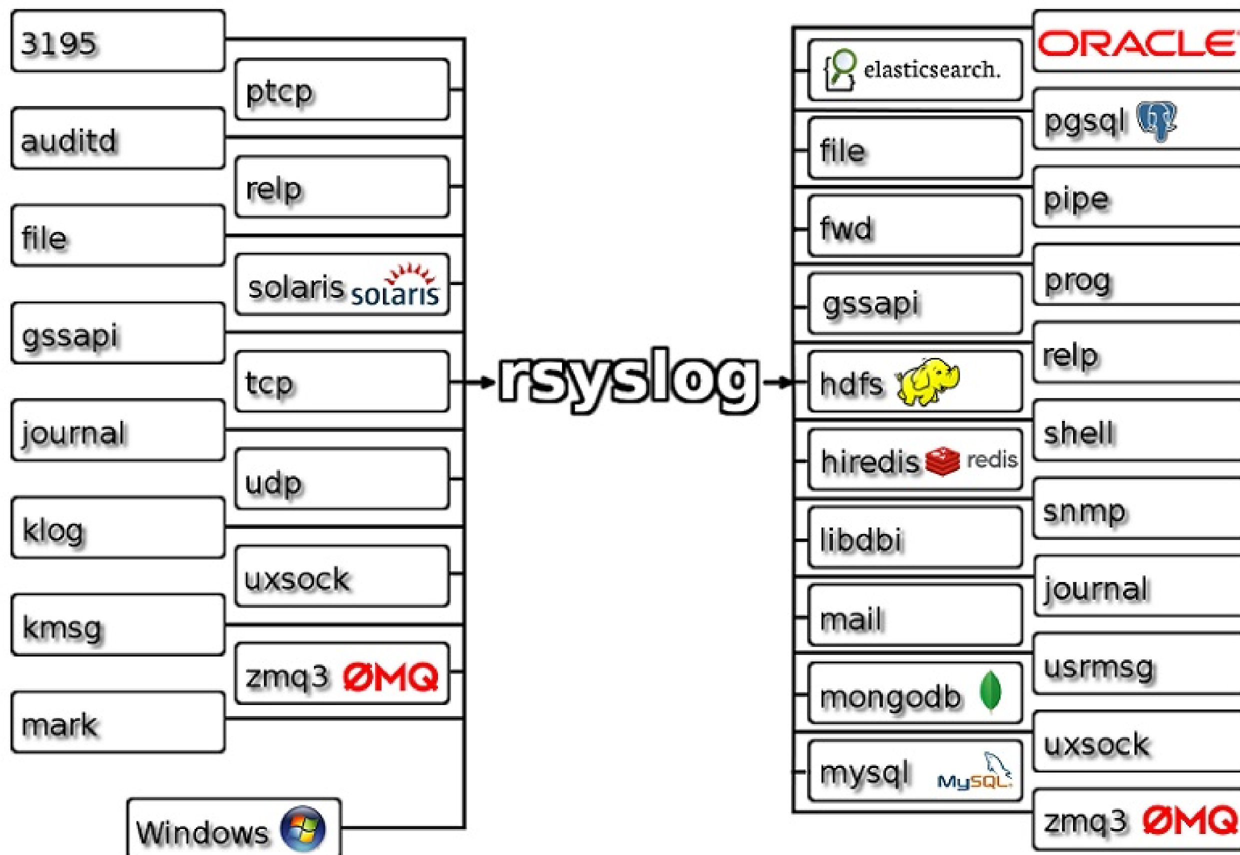
- Java版本

> java version "1.6.0" Java(TM) SE Runtime Environment (build pap6460sr14-20130705_01(SR14)) IBM J9 VM (build 2.4, JRE 1.6.0 IBM J9 2.4 AIX ppc64-64 jvmap6460sr14-20130704_155156 (JIT enabled, AOT enabled) J9VM - 20130704_155156 JIT - r9_20130517_38390 GC - GA24_Java6_SR14_20130704_1138_B155156) JCL - 20130618_01

Rsyslog

Rsyslog 是 RHEL6 开始的默认系统 syslog 应用软件(当然，RHEL 自带的版本较低，实际官方稳定版本已经到 v8 了)。官网地址：http://www.rsyslog.com

目前 Rsyslog 本身也支持多种输入输出方式，内部逻辑判断和模板处理。



常用模块介绍

不同模块插件在 rsyslog 流程中发挥作用的原理，可以阅读：http://www.rsyslog.com/doc/master/configuration/modules/workflow.html

流程中可以使用 mmnormalize 组件来完成数据的切分(相当于 logstash 的 *filters/grok* 功能)。

rsyslog 从 v7 版本开始带有 *omelasticsearch* 插件可以直接写入数据到 elasticsearch 集群，配合 mmnormalize 的使用示例见: http://puppetlabs.com/blog/use-rsyslog-and-elasticsearch-powerful-log-aggregation

而 normalize 语法说明见: http://www.liblognorm.com/files/manual/index.html?sampledatabase.htm

类似的还有 mmfields 和 mmjsonparse 组件。注意，mmjsonparse 要求被解析的 MSG 必须以 `@CEE:` 开头，解析之后的字符串为 JSON。使用示例见:http://blog.sematext.com/2013/05/28/structured-logging-with-rsyslog-and-elasticsearch/

此外，rsyslog 从 v6 版本开始，设计了一套 rainerscript 作为配置中的 DSL。利用 rainerscript 中的函数，也可以做到一些数据解析和逻辑判断：

- tolower
- cstr
- cnum
- wrap
- replace
- field
- re_extract
- re_match
- contains
- if-else
- foreach
- lookup
- set/reset/unset

详细说明见：http://www.rsyslog.com/doc/v8-stable/rainerscript/functions.html

rsyslog 与 logstash 合作

虽然 Rsyslog 很早就支持直接输出数据给 elasticsearch，但如果你使用的是 **v8.4** 以下的版本，我们这里并不推荐这种方式。因为normalize 语法还是比较简单，只支持时间，字符串，数字，ip 地址等几种。在复杂条件下远比不上完整的正则引擎。

那么，怎么使用 rsyslog 作为日志收集和传输组件，来配合 logstash 工作呢？

如果只是简单的 syslog 数据，直接单个 logstash 运行即可，配置方式见本书 2.4 章节。

如果你运行着一个高负荷运行的 rsyslog 系统，每秒传输的数据远大过单个 logstash 能处理的能力，你可以运行多个 logstash 在多个端口，然后让 rsyslog 做轮训转发(事实上，单个 omfwd 本身的转发能力也有限，所以推荐这种做法)：

```
Ruleset( name="forwardRuleSet" ) {
    Action ( type="mmsequence" mode="instance" from="0" to="4" v
ar="$.seq" )
    if $.seq == "0" then {
        action  (type="omfwd" Target="127.0.0.1" Port="5140" Pro
tocol="tcp" queue.size="150000" queue.dequeuebatchsize="2000" )
    }
    if $.seq == "1" then {
        action  (type="omfwd" Target="127.0.0.1" Port="5141" Pro
tocol="tcp" queue.size="150000" queue.dequeuebatchsize="2000" )
    }
    if $.seq == "2" then {
        action  (type="omfwd" Target="127.0.0.1" Port="5142" Pro
tocol="tcp" queue.size="150000" queue.dequeuebatchsize="2000" )
    }
    if $.seq == "3" then {
        action  (type="omfwd" Target="127.0.0.1" Port="5143" Pro
tocol="tcp" queue.size="150000" queue.dequeuebatchsize="2000" )
    }
}
```

如果 rsyslog 仅是作为 shipper 角色运行，环境中有单独的消息队列可用，rsyslog 也有对应的 omkafka, omredis, omzmq 插件可用。

rsyslog v8 版的 mmexternal 模块

如果你使用的是 v8.4 及以上版本的 rsyslog，其中有一个新加入的 mmexternal 模块。该模块是在 v7 的 omprog 模块基础上发展出来的，可以让你使用任意脚本，接收标准输入，自行处理以后再输出回来，而 rsyslog 接收到这个输出再进行下一

步处理，这就解决了前面提到的“normalize 语法太简单”的问题！

下面是使用 rsyslog 的 mmexternal 和 omelasticsearch 完成 Nginx 访问日志直接解析存储的配置。

rsyslog 配置如下：

```
module(load="imuxsock" SysSock.RateLimit.Interval="0")
module(load="mmexternal")
module(load="omelasticsearch")
template(name="logstash-index" type="list") {
    constant(value="logstash-")
    property(name="timereported" dateFormat="rfc3339" position.f
rom="1" position.to="4")
    constant(value=".")
    property(name="timereported" dateFormat="rfc3339" position.f
rom="6" position.to="7")
    constant(value=".")
    property(name="timereported" dateFormat="rfc3339" position.f
rom="9" position.to="10")
}
template( name="nginx-log" type="string" string="%msg%\n" )
if ( $syslogfacility-text == 'local6' and $programname startswit
h 'wb-www-access-' and not ($msg contains '/2/remind/unread_coun
t' or $msg contains '/2/remind/group_unread') ) then
{
    action( type="mmexternal" binary="/usr/local/bin/rsyslog-ngi
nx-elasticsearch.py" interface.input="fulljson" forcesingleinsta
nce="on" )
    action( type="omelasticsearch"
            template="nginx-log"
            server="eshost.example.com"
            bulkmode="on"
            dynSearchIndex="on"
            searchIndex="logstash-index"
            searchType="nginxaccess"
            queue.type="linkedlist"
            queue.size="50000"
            queue.dequeuebatchsize="5000"
            queue.dequeueslowdown="100000"
    )
    stop
}
```

其中调用的 python 脚本示例如下(注意只是做示例，脚本中的 split 功能其实可以用 rsyslog 的 mmfields 插件完成)：

```
#! /usr/bin/python
import sys
import json
import datetime

def nginxLog(data):
    hostname = data['hostname']
    logline = data['msg']
    time_local, http_x_up_calling_line_id, request, http_user_agent, staTus, remote_addr, http_x_log_uid, http_referer, request_time, body_bytes_sent, http_x_forwarded_proto, http_x_forwarded_for, request_uid, http_host, http_cookie, upstream_response_time = logline.split('`')
    try:
        upstream_response_time = float(upstream_response_time)
    except:
        upstream_response_time = None

    method, uri, verb = request.split(' ')
    arg = {}
    try:
        url_path, url_args = uri.split('?')
        for args in url_args.split('&'):
            k, v = args.split('=')
            arg[k] = v
    except:
        url_path = uri

    # Why %z do not implement?
    ret = {
        "@timestamp": datetime.datetime.strptime(time_local, ' [%d/%b/%Y:%H:%M:%S +0800]').strftime('%FT%T+0800'),
        "host": hostname,
        "method": method.lstrip('"'),
        "url_path": url_path,
        "url_args": arg,
        "verb": verb.rstrip('"'),
        "http_x_up_calling_line_id": http_x_up_calling_line_id,
        "http_user_agent": http_user_agent,
```

```
        "status": int(staTus),
        "remote_addr": remote_addr.strip('[]'),
        "http_x_log_uid": http_x_log_uid,
        "http_referer": http_referer,
        "request_time": float(request_time),
        "body_bytes_sent": int(body_bytes_sent),
        "http_x_forwarded_proto": http_x_forwarded_proto,
        "http_x_forwarded_for": http_x_forwarded_for,
        "request_uid": request_uid,
        "http_host": http_host,
        "http_cookie": http_cookie,
        "upstream_response_time": upstream_response_time
    }
    return ret

def onInit():
    """ Do everything that is needed to initialize processing
    """


def onReceive(msg):
    data = json.loads(msg)
    ret  = nginxLog(data)
    print json.dumps({'msg': ret})

def onExit():
    """ Do everything that is needed to finish processing. This
is being called immediately before exiting.
    """
    # most often, nothing to do here

onInit()
keepRunning = 1
while keepRunning == 1:
    msg = sys.stdin.readline()
    if msg:
        msg = msg[:len(msg)-1]
        onReceive(msg)
        sys.stdout.flush()
    else:
        keepRunning = 0
```

```
onExit()
sys.stdout.flush()
```

注意输出的时候，顶层的 key 是不能变的，msg 还得叫 msg，如果是 hostname 还得叫 hostname ，等等。否则，rsyslog 会当做处理无效，直接传递原有数据内容给下一步。

慎用提示

mmexternal 是基于 direct mode 的，所以如果你发送的数据量较大时，rsyslog 并不会像 linkedlist mode 那样缓冲在磁盘队列上，而是持续 **fork** 出新的 **mmexternal** 程序，几千个进程后，你的服务器就挂了！！所以，务必开启 `forcesingleinstance` 选项。

rsyslog 的 mmgrok 模块

Rsyslog 8.15.0 开始，附带了 mmgrok 模块，系笔者贡献。利用该模块，可以将 Logstash 的 Grok 规则，运用在 Rsyslog 中。欢迎有兴趣的读者试用。

nxlog

本节作者：松涛

nxlog 是用 C 语言写的一个跨平台日志收集处理软件。其内部支持使用 Perl 正则和语法来进行数据结构化和逻辑判断操作。不过，其最常用的场景。是在 windows 服务器上，作为 logstash 的替代品运行。

nxlog 的 windows 安装文件下载 url 见：
http://nxlog.org/system/files/products/files/1/nxlog-ce-2.8.1248.msi

配置

Nxlog默认配置文件位置在： `C:\Program Files (x86)\nxlog` 。

配置文件中有3个关键设置，分别是：input（日志输入端）、output（日志输出端）、route（绑定某输入到具体某输出）。。

例子

假设我们有两台服务器，收集其中的 windows 事务日志：

- logstash服务器ip地址：192.168.1.100
- windows测试服务器ip地址：192.168.1.101

收集流程：

1. nxlog 使用模块 im_file 收集日志文件，开启位置记录功能
2. nxlog 使用模块tcp输出日志
3. logstash 使用input/tcp ，收集日志，输出至es

Logstash配置文件

```
input {
    tcp {
        port => 514
    }
}
output{
    elasticsearch {
        host => "127.0.0.1"
        port => "9200"
        protocol => "http"
    }
}
```

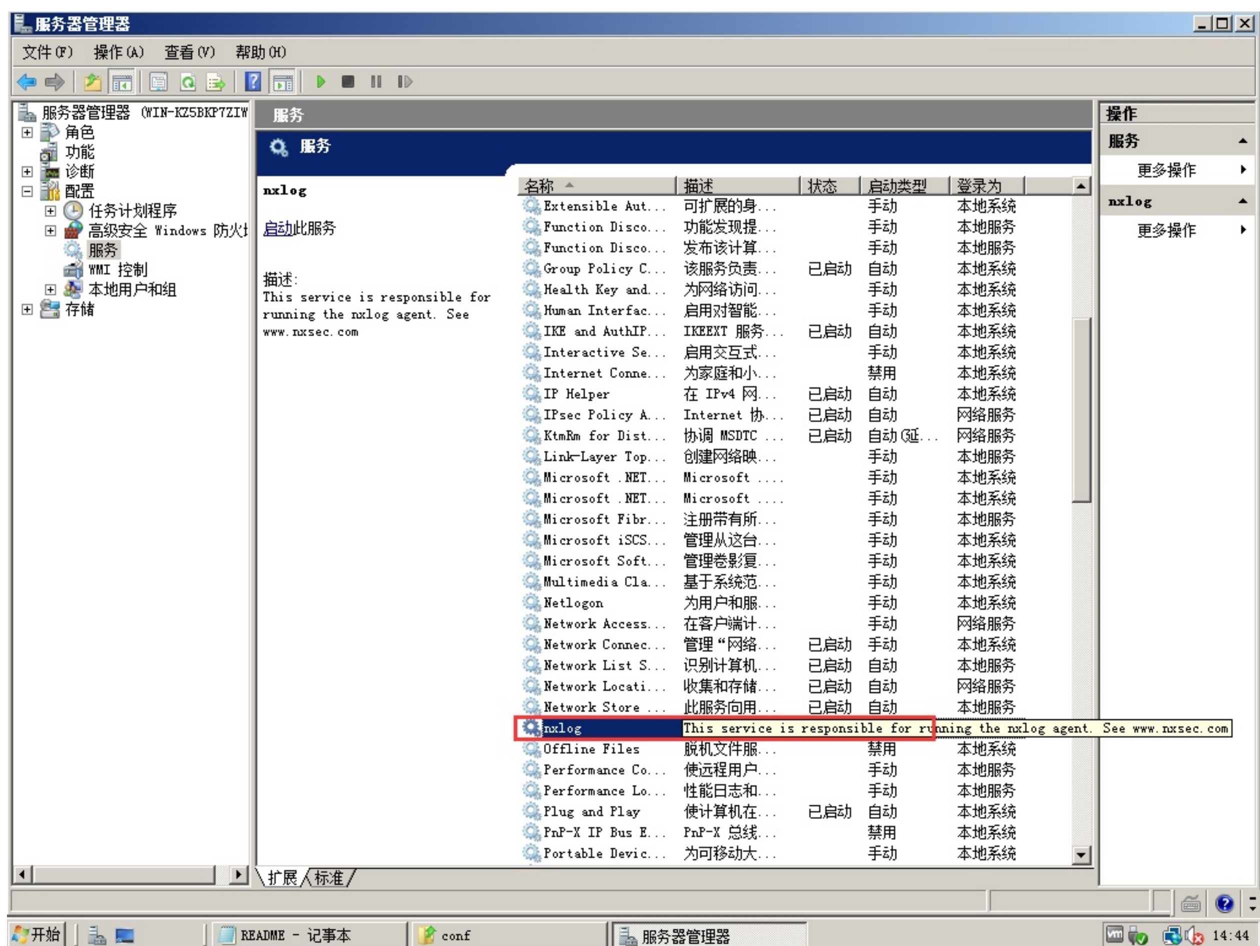
Nxlog配置文件

```
<Input testfile>
    Module im_file
    File "C:\\test\\\*.log"
    SavePos TRUE
</Input>

<Output out>
    Module om_tcp
    Host   192.168.1.100
    Port   514
</Output>

<Route 1>
    Path testfile => out
</Route>
```

配置文件修改完毕后，重启服务即可：


服务器管理器
文件(F) 操作(A) 查看(V) 帮助(H)
服务器管理器 (WIN-KZ5BKP7ZIW
角色
功能
诊断
配置
任务计划程序
高级安全 Windows 防火墙
服务
WMI 控制
本地用户和组
存储
服务
nxlog
启动此服务
描述:
This service is responsible for running the nxlog agent. See www.nxsec.com
名称 描述 状态 启动类型 登录为
Extensible Aut... 可扩展的身... 手动 本地系统
Function Disco... 功能发现提... 手动 本地服务
Function Disco... 发布该计算... 手动 本地服务
Group Policy C... 该服务负责... 已启动 自动 本地系统
Health Key and... 为网络访问... 手动 本地系统
Human Interfac... 启用对智能... 手动 本地系统
IKE and AuthIP... IKEEXT 服务... 已启动 自动 本地系统
Interactive Se... 启用交互式... 手动 本地系统
Internet Conne... 为家庭和小... 禁用 本地系统
IP Helper 在 IPv4 网... 已启动 自动 本地系统
IPsec Policy A... Internet 协... 已启动 自动 网络服务
KtmRm for Dist... 协调 MSDTC ... 已启动 自动(延... 网络服务
Link-Layer Top... 创建网络映... 手动 本地服务
Microsoft .NET... Microsoft .... 手动 本地系统
Microsoft .NET... Microsoft .... 手动 本地系统
Microsoft Fibr... 注册带有所... 手动 本地服务
Microsoft iSCS... 管理从这台... 手动 本地系统
Microsoft Soft... 管理卷影复... 手动 本地系统
Multimedia Cla... 基于系统范... 手动 本地系统
Netlogon 为用户和服... 手动 本地系统
Network Access... 在客户端计... 手动 网络服务
Network Connec... 管理"网络... 已启动 手动 本地系统
Network List S... 识别计算机... 已启动 自动 本地服务
Network Locati... 收集和存储... 已启动 自动 网络服务
Network Store ... 此服务向用... 已启动 自动 本地服务
nxlog This service is responsible for running the nxlog agent. See www.nxsec.com
Offline Files 脱机文件服... 禁用 本地系统
Performance Co... 使远程用户... 手动 本地服务
Performance Lo... 性能日志和... 手动 本地服务
Plug and Play 使计算机在... 已启动 自动 本地系统
PnP-X IP Bus E... PnP-X 总线... 禁用 本地系统
Portable Devic... 为可移动大... 手动 本地系统
扩展 标准
操作
服务
更多操作
nxlog
更多操作
开始
README - 记事本
conf
服务器管理器
14:44

Heka

heka 是 Mozilla 公司仿造 logstash 设计，用 Golang 重写的一个开源项目。同样采用了input -> decoder -> filter -> encoder -> output 的流程概念。其特点在于，在中间的 decoder/filter/encoder 部分，设计了 sandbox 概念，可以采用内嵌 lua 脚本做这一部分的工作，降低了全程使用静态 Golang 编写的难度。此外，其 filter 阶段还提供了一些监控和统计报警功能。

官网地址见：http://hekad.readthedocs.org/

Mozilla 员工已经在 *2016* 年中宣布放弃对 *heka* 项目的维护，但是社区依然坚持推动了部分代码更新和新版发布。所以本书继续保留 *heka* 的使用介绍。

下面是同样的处理逻辑，通过 syslog 接收 nginx 访问日志，解析并存储进 Elasticsearch，heka 配置文件如下：

```
[hekad]
maxprocs = 48

[TcpInput]
address = ":514"
parser_type = "token"
decoder = "shipped-nginx-decoder"

[shipped-nginx-decoder]
type = "MultiDecoder"
subs = ['RsyslogDecoder', 'nginx-access-decoder']
cascade_strategy = "all"
log_sub_errors = true

[RsyslogDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/rsyslog.lua"
    [RsyslogDecoder.config]
    type = "nginx.access"
    template = '<%pri%>%TIMESTAMP% %HOSTNAME% %syslogtag%%msg:::
sp-if-no-1st-sp%%msg:::drop-last-lf%\n'
    tz = "Asia/Shanghai"
```

```
[nginx-access-decoder]
type = "SandboxDecoder"
filename = "lua_decoders/nginx_access.lua"

    [nginx-access-decoder.config]
    type = "combined"
    user_agent_transform = true
    log_format = '[$time_local]`$http_x_up_calling_line_id`"$req
uest"`"$http_user_agent"`$staTus`[$remote_addr]`$http_x_log_uid`
"$http_referer"`$request_time`$body_bytes_sent`$http_x_forwarded
_proto`$http_x_forwarded_for`$request_uid`$http_host`$http_cooki
e`$upstream_response_time'

[ESLogstashV0Encoder]
es_index_from_timestamp = true
fields = ["Timestamp", "Payload", "Hostname", "Fields"]
type_name = "%{Type}"

[ElasticSearchOutput]
message_matcher = "Type == 'nginx.access'"
server = "http://eshost.example.com:9200"
encoder = "ESLogstashV0Encoder"
flush_interval = 50
flush_count = 5000
```

heka 目前仿造的还是旧版本的 logstash schema 设计，所有切分字段都存储在 `@fields` 下。

经测试，其处理性能跟开启了多线程 filters 的 logstash 进程类似，都在每秒 30000 条。

fluentd

Fluentd 是另一个 Ruby 语言编写的日志收集系统。和 Logstash 不同的是，Fluentd 是基于 MRI 实现的，并不是利用多线程，而是利用事件驱动。

Fluentd 的开发和使用者，大多集中在日本。

配置示例

Fluentd 受 Scribe 影响颇深，包括节点间传输采用磁盘 buffer 来保证数据不丢失等的设计，也包括配置语法。下面是一段配置示例：

```
<source>
  type tail
  read_from_head true
  path /var/lib/docker/containers/*/*-json.log
  pos_file /var/log/fluentd-docker.pos
  time_format %Y-%m-%dT%H:%M:%S
  tag docker.*
  format json
</source>
# Using filter to add container IDs to each event
<filter docker.var.lib.docker.containers.*.*.log>
  type record_transformer
  <record>
    container_id ${tag_parts[5]}
  </record>
</filter>

<match docker.var.lib.docker.containers.*.*.log>
  type copy
  <store>
    # for debug (see /var/log/td-agent.log)
    type stdout
  </store>
  <store>
    type elasticsearch
    logstash_format true
    host "#{ENV['ES_PORT_9200_TCP_ADDR']}" # dynamically configu
red to use Docker's link feature
    port 9200
    flush_interval 5s
  </store>
</match>
```

注意，虽然示例中演示的是 tail 方式。Fluentd 对应用日志，并不推荐如此读取。FLuentd 为各种编程语言提供了客户端库，应用可以直接加载日志库发送日志。下面是一个 PHP 应用的示例：

```
<?php
require_once __DIR__.'/src/Fluent/Autoloader.php';
use Fluent\Logger\FluentLogger;
Fluent\Autoloader::register();
$logger = new FluentLogger("unix:///var/run/td-agent/td-agent.so
ck");
$logger->post("fluentd.test.follow", array("from"=>"userA", "to"
=>"userB"));
```

Fluentd 使用如下配置接收即可：

```
<source>
  type unix
  path /var/run/td-agent/td-agent.sock
</source>
<match fluentd.test.**>
  type forward
  send_timeout 60s
  recover_wait 10s
  heartbeat_interval 1s
  phi_threshold 16
  hard_timeout 60s
  <server>
    name myserver1
    host 192.168.1.3
    port 24224
    weight 60
  </server>
  <server>
    name myserver2
    host 192.168.1.4
    port 24224
    weight 60
  </server>
  <secondary>
    type file
    path /var/log/fluent/forward-failed
  </secondary>
</match>
```

fluentd 插件

作为用动态语言编写的软件，fluentd 也拥有大量插件。每个插件都以 RubyGem 形式独立存在。事实上，logstash 在这方面就是学习 fluentd 的。安装方式如下：

```
/usr/sbin/td-agent-gem install fluent-plugin-elasticsearch fluen
t-plugin-grok_parser
```

fluentd 插件列表，见：http://www.fluentd.org/plugins。

Message::Passing

Message::Passing 是一个仿造 Logstash 写的 Perl5 项目。项目早期甚至就直接照原样也叫 "Logstash" 的名字。

但实际上，Message::Passing 内部原理设计还是有所偏差的。这个项目整个基于 AnyEvent 事件驱动开发框架(即著名的 libev 库)完成，也要求所有插件不要采取阻塞式编程。所以，虽然项目开发不太活跃，插件接口不甚完善，但是性能方面却非常棒。这也是我从多个 Perl 日志处理框架中选择介绍这个的原因。

Message::Passing 有比较全的 input 和 output 插件，这意味着它可以通过多种协议跟 logstash 混跑，不过 filter 插件比较缺乏。对等于 grok 的插件叫 `Message::Passing::Filter::Regexp` (我写的，嘿嘿)。下面是一个完整的配置示例：

```
use Message::Passing::DSL;
run_message_server message_chain {
    output stdout => (
        class => 'STDOUT',
    );
    output elasticsearch => (
        class => 'ElasticSearch',
        elasticsearch_servers => ['127.0.0.1:9200'],
    );
    encoder("encoder",
        class => 'JSON',
        output_to => 'stdout',
        output_to => 'es',
    );
    filter regexp => (
        class => 'Regexp',
        format => ':nginxaccesslog',
        capture => [qw( ts status remotehost url oh responsetime
 upstreamtime bytes )]
        output_to => 'encoder',
    );
    filter logstash => (
        class => 'ToLogstash',
        output_to => 'regexp',
    );
    decoder decoder => (
        class => 'JSON',
        output_to => 'logstash',
    );
    input file => (
        class => 'FileTail',
        output_to => 'decoder',
    );
};
```

源码解析

Logstash 和过去很多日志收集系统比，优势就在于其源码是用 Ruby 写的，所以插件开发相当容易。现在已经有两百多个插件可供选择。但是，随之而来的问题就是：大多数框架都用 Java 写，毕竟做大规模系统 Java 有天生优势。而另一个新生代 fluentd 则是标准的 Ruby 产品(即 Matz's Ruby Interpreter)。logstash 为什么选用 JRuby 来实现，似乎有点两头不讨好啊？

乔丹西塞曾经多次著文聊过这个问题。为了避凑字数的嫌，这里罗列他的 gist 地址：

- Time sucks 一文是关于 Time 对象的性能测试，最快的生成方法是 `sprintf` 方法，MRI 性能为 82600 call/sec，JRuby1.6.7 为 131000 call/sec，而 JRuby1.7.0 为 215000 call/sec。
- Comparing egexp patterns speeds 一文是关于正则表达式的性能测试，使用的正则统一为 `(?-mix:('(?:[^\\']+|(?:\\.)+)*'))` ，结果 MRI1.9.2 为 530000 matches/sec，而 JRuby1.6.5 为 690000 matches/sec。
- Logstash performance under ruby一文是关于 logstash 本身数据流转性能的测试，使用 *inputs/generator* 插件生成数据，*outputs/stdout* 到 pv 工具记点统计。结果 MRI1.9.3 为 4000 events/sec，而 JRuby1.7.0 为 25000 events/sec。

可能你已经运行着 logstash 并发现自己的线上数据远超过这个测试——这是因为乔丹西塞在2013年之前，一直是业余时间开发 logstash，而且从未用在自己线上过。所以当时的很多测试是在他自己电脑上完成的。

在 logstash 得到大家强烈关注后，作者发表了《logstash needs full time love》，表明了这点并求一份可以让自己全职开发 logstash 的工作，同时列出了1.1.0 版本以后的 roadmap。（不过事实证明当时作者列出来的这些需求其实不紧急，因为大多数，或者说除了 kibana 以外，至今依然没有==!）

时间轴继续向前推，到 2011 年，你会发现 logstash 原先其实也是用 MRI1.8.7 写的！在 grok 模块从 C 扩展改写成 FFI 扩展后，才正式改用 JRuby。

切换语言的当时，乔丹西塞发表了《logstash, why jruby?》大家可以一读。

事实上，时至今日，多种 Ruby 实现的痕迹(到处都有 RUBY_ENGINE 变量判断)依然遍布 logstash 代码各处，作者也力图保证尽可能多的代码能在 MRI 上运行。

作为简单的提示，在和插件无关的核心代码中，只有 LogStash::Event 里生成 `@timestamp` 字段时用了 Java 的 joda 库为 JRuby 仅有的。稍微修改成 Ruby 自带的 Time 库，即可在 MRI 上运行起来。而主要插件中，也只有 filters/date 和 outputs/elasticsearch 是 Java 相关的。

另一个温馨预警，Logstash 被 Elastic.co 收购以后，又有另一种讨论中的发展方向，就是把 logstash 的 core 部分代码，尽量的 JVM 通用化，未来，可以用 JRuby，Jython、Scala，Groovy，Clojure 和 Java 等任意 JVM 平台语言写 Logstash 插件。

这就是开源软件的多样性未来，让我们拭目以待吧~

pipeline

在一开始，就介绍过，Logstash 对日志的处理，从 input 到 output，就像在 Linux 命令行上的管道操作一样。事实上，在 Logstash 中，对此有一个专门的名词，叫 Pipeline。

Pipeline 的代码加载路径如下：

> `bin/logstash` -> `logstash-core/lib/logstash/runner.rb` -> `logstash-core/lib/logstash/agent.rb` -> `logstash-core/lib/logstash/pipeline.rb`

Logstash 从 2.2 版开始对 pipeline 做了大幅重构，目前最新 5.0 版的 `pipeline.rb`，可以归纳成下面这么一段缩略版的代码：

```
    # 初始化阶段
    @config = grammar.parse(configstr)
    code = @config.compile
    eval(code)

    queue = LogStash::Util::WrappedSynchronousQueue.new
    @input_queue_client = queue.write_client
    @filter_queue_client = queue.read_client

    # 启动指标计数器
    @filter_queue_client.set_events_metric()
    @filter_queue_client.set_pipeline_metric()

    # 运行
    LogStash::Util.set_thread_name("[#{pipeline_id}]-pipeline-ma
nager")

    # 启动输入插件
    @inputs.each do |input|
        input.register
        @input_threads << Thread.new do
            LogStash::Util::set_thread_name("[#{pipeline_id}]<#{
input.class.config_name}")
```

```
            plugin.run(@input_queue)
        end
    end

    @outputs.each {|o| o.register }
    @filters.each {|f| f.register }

    max_inflight = batch_size * pipeline_workers
    pipeline_workers.times do |t|
        @worker_threads << Thread.new do
            LogStash::Util.set_thread_name("[#{pipeline_id}]>wor
ker#{t}")
            @filter_queue_client.set_batch_dimensions(batch_size
, batch_delay)
            while true
                batch = @filter_queue_client.take_batch

                # 开始过滤
                batch.each do |event|
                    filter_func(event).each do |e|
                        batch.merge(e)
                    end
                end
                # 计数
                @filter_queue_client.add_filtered_metrics(batch)

                # 开始输出
                output_events_map = Hash.new { |h,k|  h[k] = []
}
                batch.each do |event|
                    output_func(event).each do |output|
                        output_events_map[output].push(event)
                    end
                end
                output_events_map.each do |output, events|
                    output.multi_receive(events)
                end
                @filter_queue_client.add_output_metrics(batch)

                # 释放
```

```
                    @filter_queue_client.close_batch(batch)
                end
            end
        end

        # 运行
        @input_threads.each(&:join)
```

整个缩略版，可以了解到一个关键信息，对我们理解 Logstash 原理是最有用的： `queue` 是一个固定大小为 0 的多线程同步队列。filter 和 output 插件，则在相同的 `pipeline_worker` 线程中运行，该线程每次批量获取数据，也批量传递给 filter 和 output 插件。

由于 input 到 filter 之间有唯一的队列，任意一个 filter 或者 output 发生堵塞，都会一直堵塞到最前端的接收。这也是 logstash-input-heartbeat 的理论基础。

这个全新的 NG pipeline 是从 2.2 版开始发布的，当时也导致 logstash-output-elasticsearch 的 ESClient 数量比过去大幅增加，对写入 Elasticsearch 的性能是不利的。随后官方意识到这个问题，并大举重构了 logstash-output-elasticsearch 的实现，改成了一个整体连接池的方式，代码见：https://github.com/logstash-plugins/logstash-output-elasticsearch/commit/06a47535111881b2bc6c9dbd3908e664e4852476。相关的新配置参数，在之前插件介绍中已经讲过。

Logstash 中 Event 的生成

上一节大家可能注意到了，整个 pipeline 非常简单，无非就是一个多线程的线程间数据读写。但是，之前介绍的 codec 在哪里？这个问题，并不在 pipeline 中完成，而是 plugin 中。

Logstash 从 1.5 开始，把各个 plugin 拆分成了单独的 gem，主代码里只留下了几个 `base.rb` 类。所以，要了解详细情况，我们需要阅读一个实际跑数据的插件，比如 `vendor/bundle/jruby/1.9/gems/logstash-input-stdin-3.2.0/lib/logstash/inputs/stdin.rb` 。

可以看到其中最关键的读取数据部分代码如下：

```
    @host = Socket.gethostname
    while !stop?
      if data = stdin_read
      @codec.decode(data) do |event|
        decorate(event)
        event.set("host", @host) if !event.include?("host")
        queue << event
      end
    end
```

这里两个关键函数： `@codec.decode(line)` 和 `decorate(event)` 。

@codec 在 `stdin.rb` 中默认为 line，那么我们就继续看 `vendor/bundle/jruby/1.9/gems/logstash-codec-line-3.0.2/lib/logstash/codecs/line.rb` 的相关部分：

```
  def register
    require "logstash/util/buftok"
    @buffer = FileWatch::BufferedTokenizer.new(@delimiter)
    @converter = LogStash::Util::Charset.new(@charset)
    @converter.logger = @logger
  end
  public
  def decode(data)
    @buffer.extract(data).each do |line|
      yield LogStash::Event.new("message" => @converter.convert(
line))
    end
  end # def decode
```

超简短。就是在这个 `@codec.decode(data)` 里，生成了 `LogStash::Event` 对象。那么，我们通过 `output { codec => rubydebug }` 看到的除了 message 字段以外的那些数据，又是怎么来的呢？尤其是那个 `@timestamp` 是怎么出来的？

在 5.0 之前，我们可以通过 `lib/logstash/event.rb` 看到相关属性的定义和操作。5.0 之后，Logstash 为了提高性能，对 Event 部分采用 Java 语言进行了重构，现在你在 `logstash-core-event-java/lib/logstash/event.rb` 里只能看到通过 JRuby 的专属 require 指令加载 jar 的语句了。

想要了解 Logstash::Event 的实际定义，需要去 Git 仓库下载，然后阅读 Java 源代码，你也可以直接通过网页阅读，地址是：https://github.com/elastic/logstash/blob/master/logstash-core-event-java/src/main/java/org/logstash/Event.java ：

```
  public static final String METADATA = "@metadata";
  public static final String METADATA_BRACKETS = "[" + METADATA
+ "]";
  public static final String TIMESTAMP = "@timestamp";
  public static final String TIMESTAMP_FAILURE_TAG = "_timestamp
parsefailure";
  public static final String TIMESTAMP_FAILURE_FIELD = "_@timest
amp";
  public static final String VERSION = "@version";
  public static final String VERSION_ONE = "1";

  public Event()
  {
      this.metadata = new HashMap<String, Object>();
      this.data = new HashMap<String, Object>();
      this.data.put(VERSION, VERSION_ONE);
      this.cancelled = false;
      this.timestamp = new Timestamp();
      this.data.put(TIMESTAMP, this.timestamp);
      this.accessors = new Accessors(this.data);
      this.metadata_accessors = new Accessors(this.metadata);
  }
```

现在就清楚了，这个特殊的 `@timestamp` 是在 event 对象初始化的时候加上的，其实现同样在这个 Java 源码中，见 https://github.com/elastic/logstash/blob/master/logstash-core-event-java/src/main/java/org/logstash/Timestamp.java：

```
  public class Timestamp implements Cloneable {
    private DateTime time;
    public Timestamp() {
      this.time = new DateTime(DateTimeZone.UTC);
    }
  }
```

这就是我们看到 Logstash 生成的事件总是 UTC 时区时间的原因。

至于如果一开始就传入了 `@timestamp` 数据的处理，则是这样：

```
  public Timestamp(String iso8601) {
    this.time = ISODateTimeFormat.dateTimeParser().parseDateTime
(iso8601).toDateTime(DateTimeZone.UTC);
  }
  public Timestamp(long epoch_milliseconds) {
    this.time = new DateTime(epoch_milliseconds, DateTimeZone.UT
C);
  }
```

同样会利用 joda 库做一次解析，还是转换成 UTC 时区。

自己写一个插件

前面已经提过在运行 logstash 的时候，可以通过 `--pluginpath` 参数来加载自己写的插件。那么，插件又该怎么写呢？

插件格式

一个标准的 logstash 输入插件格式如下：

```
require 'logstash/namespace'
require 'logstash/inputs/base'
class LogStash::Inputs::MyPlugin < LogStash::Inputs::Base
  config_name 'myplugin'
  default :codec, "line"
  config :myoption_key, :validate => :string, :default => 'myopt
ion_value'
  public def register
  end
  public def run(queue)
  end
end
```

其中大多数语句在过滤器和输出阶段是共有的。

- config_name 用来定义该插件写在 logstash 配置文件里的名字；
- config 可以定义很多个，即该插件在 logstash 配置文件中的可配置参数。logstash 很温馨的提供了验证方法，确保接收的数据是你期望的数据类型；
- register logstash 在启动的时候运行的函数，一些需要常驻内存的数据，可以在这一步先完成。比如对象初始化，*filters/ruby* 插件中的 `init` 语句等。

插件的关键方法

输入插件独有的是 run 方法。在 run 方法中，必须实现一个长期运行的程序(最简单的就是 loop 指令)。然后在每次收到数据并处理成 `event` ，这段示例在上一节演示 Logstash::Event 的生成时已经介绍过。最后，一定要调用 `queue << event`

语句。一个输入流程就算是完成了。

而如果是过滤器插件，对应修改成：

```
require 'logstash/filters/base'
class LogStash::Filters::MyPlugin < LogStash::Filters::Base
  config_name 'myplugin'
  public def register
  end
  public def filter(event)

    filter_matched(event)
  end
end
```

其中， `filter_matched` 是在 filter 函数完成本插件自己的处理逻辑之后一定要调用的。

输出插件则是：

```
require 'logstash/outputs/base'
class LogStash::Outputs::MyPlugin < LogStash::Outputs::Base
  config_name 'myplugin'
  concurrency :single
  public def register
  end
  public def multi_receive(events)
  end
end
```

这里和过去的版本最明显的差别是，处理方法改成了 `multi_receive` ，而不是 `receive` 。因为新的 pipeline 机制是批量传递数据给输出插件的。不过为了兼容过去的插件， `LogStash::Outputs::Base` 基类中的 `multi_receive` 实现继续迭代调用了 `receive` 。

另一个是新出现的配置 `concurrency` ，代表着本插件是否 threadsafe，并由此取代了过去的 workers 选项。可选项为： `single` 和 `shared` 。

- single 表示本插件是非线程安全的，必须在各 pipeline workers 之间同一时刻

只有一个运行。
- shared 表示本插件是线程安全的，每个 pipeline workers 之间可以独立运行，这也就意味着插件作者要自己在 `multi_receive` 里调用 Mutexes。

推荐阅读

- Extending logstash
- Plugin Milestones

插件打包

Logstash 从 1.5.0-GA 版开始，对插件规范做了重大变更。放弃了 milestone 定义，去除了 `--pluginpath` 命令行参数。统一改成 `bin/plugin` 管理的 rubygem 包插件。那么，我们自己写的 Logstash 插件，也同样需要适应这个新规则，写完 Ruby 代码，还要打包成 gem 才能使用。

为了我们更方便的完成工作，logstash 针对 4 种插件形态提供了 4 个示例库，可以按照自己所需克隆使用。比如要写一个 logstash-filter-mything 插件：

```
git clone https://github.com/logstash-plugins/logstash-filter-example
cd logstash-filter-example
mv logstash-filter-example.gemspec logstash-filter-mything.gemspec
mv lib/logstash/filters/example.rb lib/logstash/filters/mything.rb
mv spec/filters/example_spec.rb spec/filters/mything_spec.rb
```

然后把代码写在 `lib/logstash/filters/mything.rb` 里即可。

代码部分完成。然后就是定义 gem 打包需要的额外文件和库依赖了。

目录中有两个文件， `Gemfile` 和 `logstash-filter-mything.gemspec` 。

`Gemfile` 文件就是标准格式，用来运行 `bundler install` 时下载 rubygems 包的。默认情况下，最基础的内容是：

```
source 'https://rubygems.org' # 国内建议改成 'http://ruby.taobao.o
rg'
gemspec
gem "logstash", :github => "elastic/logstash", :branch => "1.5"
```

`gemspec` 文件则是用来定义软件包本身规范，不单限于 rubygems。示例如下：

```
Gem::Specification.new do |s|
  s.name = 'logstash-filter-mything'
  s.version         = '1.1.0'
  s.licenses = ['Apache License (2.0)']
  s.summary = "This mything filter is just for Elastic Stack Gui
de example"
  s.description = "This gem is a logstash plugin required to be
installed on top of the Logstash core pipeline using $LS_HOME/bi
n/logstash-plugin install gemname. This gem is not a stand-alone
 program"
  s.authors = ["Chenryn"]
  s.email = 'chenlin7@staff.sina.com.cn'
  s.homepage = "http://kibana.logstash.es"
  s.require_paths = ["lib"]

  s.files = `find . -type f ! -wholename '*.svn*'`.split($\)
  s.test_files = s.files.grep(%r{^(test|spec|features)/})

  s.metadata = { "logstash_plugin" => "true", "logstash_group" =
> "filter" }

  s.requirements << "jar 'org.elasticsearch:elasticsearch', '1.4
.0'"
  s.add_runtime_dependency "jar-dependencies"
  s.add_runtime_dependency "logstash-core", '>= 1.4.0', '< 2.0.0
'
  s.add_development_dependency 'logstash-devutils'
end
```

其中:

- `s.version` 就是 milestone 的替代品，0.1.x 相当于是 milestone 0；0.9.x 相当于是 milestone 2；1.x.x 相当于是 milestone 3。
- `s.file` 默认写法是 `git ls-files` ，因为默认是 git 库，如果你本身采用了 svn，或者 cvs 库，都不要紧，只要命令列出的是你需要打包进去的文件即可。
- `s.metadata` 是 logstash 的 plugin 命令在 install 的时候会提前 verify 的特殊信息，一定要保留。
- `s.add_runtime_dependency` 是定义插件依赖库的指令。如果有 jar 包依赖，则额外再加 `s.requirements` 。

好了，全部完毕。下面打包：

```
gem build logstash-filter-mything.gemspec
```

运行完就会生成一个 *logstash-filter-mything-1.1.0.gem* 软件包，可以安装使用了。

读取二进制文件(utmp)

本节作者：*NERO*

我们知道Linux系统有些日志不是以可读文本形式存在文件中，而是以二进制内容存放的。比如utmp等。

Files插件在读取二进制文件和文本文件的区别在于没办法一行行处理内容，内容也不是直观可见的。但是在其他处理逻辑是保持一致的，比如多文件支持、断点续传、内容监控等等，区别在于bytes的截断和bytes的转换。 在Files的基础上，新增了个插件utmp，插件实现了对二进制文件的读取和解析，实现类似于文本文件一行行输出内容，以供Filter做进一步分析。

utmp新增了两个配置项：`struct_size` 和 `struct_format` 。

`struct_size` 代表结构体的大小，number类型；`struct_format` 为结构体的内存分布格式，string类型。

配置实例

```
input {
  utmp {
    path => "/var/run/utmp"
    type => "syslog"
    start_position => "beginning"
    struct_size => 384
    struct_format => "s2Ia32a4a32a256s2iI2I4a20"
  }
}
```

我们可以通过 `man utmp` 了解 utmp 的结构体声明。通过结构体声明我们能知道结构体的内存分布情况。我们看 `struct_format => "s2Ia32a4a32a256s2iI2I4a20"` 中的第一个 **s**，s 代表 short，后面的数字 2 代表 short 元素的个数，那么意思就是结构体的第一个和第二个元素是 short 类型，各占 2 个字节。

以此类推。s2Ia32a4a32a256s2iI2I4a20 总共代表 384 个字节，与 `struct_size` 配置保持一致。s2Ia32a4a32a256s2iI2I4a20 本质上是对应于 Ruby 中的 unpack 函数的参数。其中 s、T、a、i 等等含义参见 Ruby 的 unpack 函数说明。

需要注意的是，你必须了解当前操作系统的字节序是大端模式还是小端模式，对于数字类型的结构体元素很有必要，比如对于 i 和 I 的选择。还有一个就是，结构体内存对齐的问题。实际上 utmp 结构体第一个元素 2 字节，第二个元素是 4 字节。编译器在编译的时候会在第一个 2 字节后插入 2 字节来补齐 4 字节，知道这点尤为重要，所以其实第二个 short 是无意义的，只是为了内存对其才有补的。

输出

经过 utmp 插件的 event，message 字段形如 "field1|field2|field3"，以 "|" 分隔每个结构体元素。可用Filter插件进一步去做解析。utmp 本身只负责将二级制内容转换成可读的文本，不对文件编码格式负责。

安装

插件地址：https://github.com/txyafx/logstash-input-utmp

```
git clone https://github.com/txyafx/logstash-input-utmp
cd logstash-input-utmp
rm -rf .git
vim Gemfile 修改插件绝对路径
git init
git add .
gem clean
gem build logstash-input-utmp.gemspec
```

用logstash安装本地utmp插件

Beats 平台

Beats 平台是 Elastic.co 从 packetbeat 发展出来的数据收集器系统。beat 收集器可以直接写入 Elasticsearch，也可以传输给 Logstash。其中抽象出来的 libbeat，提供了统一的数据发送方法，输入配置解析，日志记录框架等功能。

也就是说，所有的 beat 工具，在配置上，除了 input 以外，在 output、filter、shipper、logging、run-options 上的配置规则都是完全一致的。

filter

5.0 版本后，beats 新增了简单的 filter 功能，用来完成事件过滤和字段删减：

```
filters:
    - drop_event:
        regexp:
            message: "^DBG:"
    - drop_fields:
        contains:
            source: "test"
        fields: ["message"]
    - include_fields:
        fields: ["http.code", "http.host"]
        equals:
            http.code: 200
        range:
            gte:
                cpu.user_p: 0.5
            lt:
                cpu.user_p: 0.8
```

可用的条件判断包括：

- equals
- contains
- regexp

- range
- or
- and
- not

output

目前 beat 可以发送数据给 Elasticsearch、Logstash、File、Kafka、Redis 和 Console 六种目的地址。

Elasticsearch

beats 发送到 Elasticsearch 也是走 HTTP 接口。示例配置段如下：

```
output:
    elasticsearch:
        hosts: ["http://localhost:9200", "https://onesslip:9200/
path", "anotherip"]
        parameters: {pipeline: my_pipeline_id}
       # 仅用于 Elasticsearch 5.0 以后的 ingest 方式
        username: "user"
        password: "pwd"
        index: "topbeat"
        bulk_max_size: 20000
        flush_interval: 5
        tls:
            certificate_authorities: ["/etc/pki/root/ca.pem"]
            certificate: "/etc/pki/client/cert.pem"
            certificatekey: "/etc/pki/client/cert.key"
```

- `hosts` 中可以通过 URL 的不同形式，来表示 HTTP 还是 HTTPS，是否有添加代理层的 URL 路径等情况。
- `index` 表示写入 Elasticsearch 时索引的前缀，比如示例即表示索引名为 `topbeat-yyyy.MM.dd`

Logstash

beat 写入 Logstash 时，会配合 Logstash-1.5 后新增的 metadata 特性。将 beat 名和 type 名记录在 metadata 里。所以对应的 Logstash 配置应该是这样：

```
input {
    beats {
        port => 5044
    }
}
output {
    elasticsearch {
        hosts => ["http://localhost:9200"]
        index => "%{[@metadata][beat]}-%{+YYYY.MM.dd}"
        document_type => "%{[@metadata][type]}"
    }
}
```

beat 示例配置段如下:

```
output:
    logstash:
        hosts: ["localhost:5044", "localhost:5045"]
        worker: 2
        loadbalance: true
        index: topbeat
```

这里 `worker` 的含义，是 beat 连到每个 host 的线程数。在 `loadbalance` 开启的情况下，意味着有 4 个worker 轮训发送数据。

File

```
output:
    file:
        path: "/tmp/topbeat"
        filename: topbeat
        rotate_every_kb: 1000
        number_of_files: 7
```

Kafka

```
output:
    kafka:
        hosts: ["kafka1:9092", "kafka2:9092", "kafka3:9092"]
        topic: '%{[type]}'
        topics:
            - key: "info_list"
              when:
                  contains:
                      message: "INFO"
            - key: "debug_list"
              when:
                  contains:
                      message: "DEBUG"
            - key: "%{[type]}"
              mapping:
                  "http": "frontend_list"
                  "nginx": "frontend_list"
                  "mysql": "backend_list"
        partition:
            round_robin:
                reachable_only: true
        required_acks: 1
        compression: gzip
        max_message_bytes: 1000000
```

- 大于 `max_message_bytes` 长度的事件(注意不只是原日志长度)会被直接丢弃。
- partition 策略默认为 hash。可选项还有 random 和 round_robin。
- compression 可选项还有 none 和 snappy。
- required_acks 可选项有 -1、0 和 1。分别代表：等待全部副本完成、不等待、等待本地完成。
- topics 用来配置基于匹配规则的选择器，支持 when 和 mapping，when 条件下可以使用上小节列出的各种 filter。如果都匹配不上，则采用 topic 配置。

Redis

```
output:
    redis:
        hosts: ["localhost"]
        password: "my_password"
        key: "filebeat"
        db: 0
        timeout: 5
```

Redis 输出也有 keys 配置。方式和 Kafka 的 topics 类似。

Console

```
output:
    console:
        pretty: true
```

shipper

shipper 部分是一些和网络拓扑相关的配置，就目前来说，大多数是 packetbeat 独有的。

```
shipper:
    name: "my-shipper"
    tags: ["my-service", "hardware", "test"]
    ignore_outgoing: true
    refresh_topology_freq: 10
    topology_expire: 15
    geoip:
        paths:
            - "/usr/share/GeoIP/GeoLiteCity.dat"
```

logging

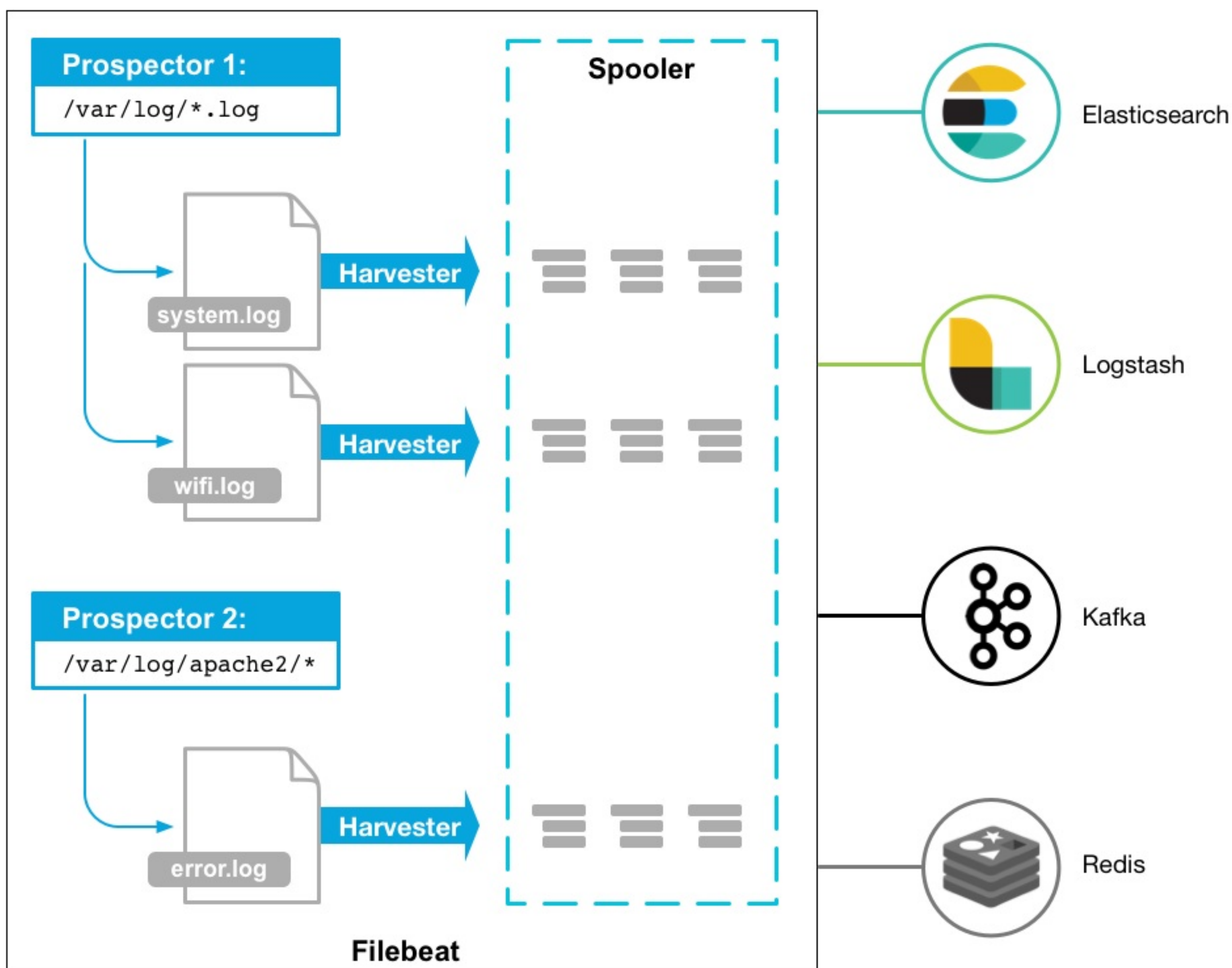
```
logging:
    level: warning
    to_files: true
    to_syslog: false
    files:
        path: /var/log/mybeat
        name: mybeat.log
        keepfiles: 7
```

run options

```
runoptions:
    uid=501
    gid=501
```

filebeat

filebeat 是基于原先 logstash-forwarder 的源码改造出来的。换句话说：filebeat 就是新版的 logstash-forwarder，也会是 Elastic Stack 在 shipper 端的第一选择。



安装部署

- deb:

```
curl -L -O https://download.elastic.co/beats/filebeat/filebeat_5.0.0_amd64.deb
sudo dpkg -i filebeat_5.0.0_amd64.deb
```

- rpm:

```
curl -L -O https://download.elastic.co/beats/filebeat/filebe
at-5.0.0-x86_64.rpm
sudo rpm -vi filebeat-5.0.0-x86_64.rpm
```

- mac:

```
curl -L -O https://download.elastic.co/beats/filebeat/filebe
at-5.0.0-darwin.tgz
tar xzvf filebeat-5.0.0-darwin.tgz
```

- win:

- 下载 https://download.elastic.co/beats/filebeat/filebeat-5.0.0-windows.zip

- 解压到 C:\Program Files
- 重命名 filebeat-5.0.0-windows 目录为 Filebeat
- 右键点击 PowerSHell 图标，选择『以管理员身份运行』
- 运行下列命令，将 Filebeat 安装成 windows 服务：

```
PS > cd 'C:\Program Files\Filebeat'
PS C:\Program Files\Filebeat> .\install-service-filebeat.ps1
```

注意

可能需要额外授予执行权限。命令为： `PowerShell.exe -ExecutionPolicy RemoteSigned -File .\install-service-filebeat.ps1` .

filebeat 配置

所有的 beats 组件在 output 方面的配置都是一致的，之前章节已经介绍过。这里只介绍 filebeat 在 input 段的配置，如下：

```
filebeat:
    spool_size: 1024                                    # 最大可
以攒够 1024 条数据一起发送出去
    idle_timeout: "5s"                                  # 否则每
5 秒钟也得发送一次
    registry_file: ".filebeat"                          # 文件读
```

```
取位置记录文件，会放在当前工作目录下。所以如果你换一个工作目录执行 filebeat 会导致重复传输！
    config_dir: "path/to/configs/contains/many/yaml"    # 如果配置过长，可以通过目录加载方式拆分配置
    prospectors:                                        # 有相同配置参数的可以归类为一个 prospector
        -
            fields:
                ownfield: "mac"                         # 类似 logstash 的 add_fields
            paths:
                - /var/log/system.log                   # 指明读取文件的位置
                - /var/log/wifi.log
            include_lines: ["^ERR", "^WARN"]            # 只发送包含这些字样的日志
            exclude_lines: ["^OK"]                      # 不发送包含这些字样的日志
        -
            document_type: "apache"                     # 定义写入 ES 时的 _type 值
            ignore_older: "24h"                         # 超过 24 小时没更新内容的文件不再监听。在 windows 上另外有一个配置叫 force_close_files，只要文件名一变化立刻关闭文件句柄，保证文件可以被删除，缺陷是可能会有日志还没读完
            scan_frequency: "10s"                       # 每 10 秒钟扫描一次目录，更新通配符匹配上的文件列表
            tail_files: false                           # 是否从文件末尾开始读取
            harvester_buffer_size: 16384                # 实际读取文件时，每次读取 16384 字节
            backoff: "1s"                               # 每 1 秒检测一次文件是否有新的一行内容需要读取
            paths:
                - "/var/log/apache/*"                   # 可以使用通配符
            exclude_files: ["/var/log/apache/error.log"]
        -
            input_type: "stdin"                         # 除了 "log"，还有 "stdin"
```

```
            multiline:                                      # 多行合并
                pattern: '^[[:space:]]'
                negate: false
                match: after
output:
    ...
```

我们已完成了配置，当 `sudo service filebeat start` 之后，你就可以在 kibana 上看到你的日志了。

字段

Filebeat 发送的日志，会包含以下字段：

- `beat.hostname` beat 运行的主机名
- `beat.name` shipper 配置段设置的 `name`，如果没设置，等于 `beat.hostname`
- `@timestamp` 读取到该行内容的时间
- `type` 通过 `document_type` 设定的内容
- `input_type` 来自 "log" 还是 "stdin"
- `source` 具体的文件名全路径
- `offset` 该行日志的起始偏移量
- `message` 日志内容
- `fields` 添加的其他固定字段都存在这个对象里面

packetbeat

目前 packetbeat 支持的网络协议有：HTTP，MySQL，PostgreSQL，Redis，Thrift，DNS，MongoDB，Memcache。

对于很多 Elastic Stack 新手来说，面对的很可能就是几种常用数据流，而书写 logstash 正则是一个耗时耗力的重复劳动，文件落地本身又是多余操作，packetbeat 的运行方式，无疑是对新手入门极大的帮助。

安装部署

packetbeat 同样有已经编译完成的软件包可以直接安装使用。需要注意的是，packetbeat 支持不同的抓包方式，也就有不同的依赖。比如最通用的 *pcap*，就要求安装有 `libpcap` 包，*pf_ring* 就要求有 `pfring` 包，而在 windows 平台上，则需要下载安装 WinPcap 软件。

```
# yum install libpcap
# rpm -ivh http://www.nmon.net/packages/rpm6/x86_64/PF_RING/pfring-6.1.1-83.x86_64.rpm
# rpm -ivh https://download.elasticsearch.org/beats/packetbeat/packetbeat-5.0.0-x86_64.rpm
```

packetbeat 还附带了一个定制的 Elasticsearch 模板，要在正式使用前导入 ES 中。

```
# curl -XPUT 'http://localhost:9200/_template/packetbeat' -d@/etc/packetbeat/packetbeat.template.json
```

配置示例

通过 RPM 安装的 packetbeat 配置文件位于 `/etc/packetbeat/packetbeat.yml`。其示例如下：

```
interfaces:
```

```
    device: any
    type: af_packet
    snaplen: 65536                              # 保证大过 MTU 即可，所以公网抓包的话可以改成 1514
    buffer_size_mb: 30                          # 该参数仅在 af_packet 时有效，表示在 linux 内核空间和用户空间之间开启多大共享内存，可以减低 CPU 消耗
    with_vlans: false                           # 在生成抓包语句的时候，是否带上 vlan 标示。示例："port 80 or port 3306 or (vlan and (port 80 or port 3306))"
protocols:
    dns:
        ports: [53]
        send_request: false                     # 通用配置：是否将原始的 request 字段内容都发给 ES
        send_response: false                    # 通用配置：是否将原始的 response 字段内容都发给 ES
        transaction_timeout: "10s"              # 通用配置：超过 10 秒的不再等待响应，直接认定为一个事务，发送给 ES
        include_additionals: false              # DNS 专属配置：是否带上 dns.additionals 字段
        include_authorities: false              # DNS 专属配置：是否带上 dns.authority 字段
    http:
        ports: [80, 8080]                       # 抓包的监听端口
        hide_keywords: ["password","passwd"]    # 对 GET 方法的请求参数，POST 方法的顶层参数里的指定关键信息脱敏。注意：如果你又开启了 send_request，request 字段里的并不会脱敏"
        redact_authorization: true              # 对 header 中的 Authorization 和 Proxy-Authorization 内容做模糊处理。道理和上一条类似
        send_headers: ["User-Agent"]            # 只在 headers 对象里记录指定的 header 字段内容
        send_all_headers: true                  # 在 headers 对象里记录所有 header 字段内容
        include_body_for: ["text/html"]         # 在开启 send_response 的前提下，只记录某些类别的响应体内容
        split_cookie: true                      # 将 headers 对象里的 Cookie 或 Set-Cookie 字段内容再继续做 KV 切割
        real_ip_header: "X-Forwarded-For"       # 将 headers 对象里的指定字段内容设为 client_location 和 real_ip 字段
```

```
    memcache:
        ports: [11211]
        parseunknown: false
        maxvalues: 0
        maxbytespervalue: 100
        udptransactiontimeout: 10000
    mysql:
        ports: [3306]
        max_rows: 10                                  # 发送给 ES 的 S
QL 内容最多为 10 行
        max_row_length: 1024                          # 发送给 ES 的 S
QL 内容每行最长为 1024 字节
    cassandra:
        send_request_header: true                     # 在开启了 send_
request 的前提下，记录 cassandra_request.request_headers 字段
        send_response_header: true
        compressor: "snappy"
        ignored_ops: ["SUPPORTED","OPTIONS"]          # 不记录部分操作
    amqp:
        ports: [5672]
        max_body_length: 1000                         # 超长的都截断掉
        parse_headers: true
        parse_arguments: false
        hide_connection_information: true             # 不记录打开、关闭
连接之类的信息
    thrift:
        transport_type: socket                        # Thrift 传输类
型，"socket" 或 "framed"
        protocol_type: binary
        idl_files: ["shared.thrift"]                  # 一般来说默认内置
的 IDL 文件已经足够了
        string_max_size: 200                          # 参数或返回值中字
符串的最大长度，超长的都截断掉
        collection_max_size: 15                       # Thrift 的 lis
t, map, set, structure 结构中的元素个数上限，超过的元素丢弃
        capture_reply: true                           # 为了节省资源，可
以设置 false，只解析响应中的方法名称，其余信息丢弃
        obfuscate_strings: true                       # 把所有方法参数、
返回码、异常结构中的字符串都用 * 代替
        drop_after_n_struct_fields: 500               # 在 packetbeat
```

```
结束整个事务之前，一个结构体最多能存多少个字段。该配置主要考虑节约内存
    monogodb:
        max_docs: 10                                     # 设置 send_resp
onse 前提下，response 字段最多保存多少个文档。超长的都截断掉。不限则设为 0
        max_doc_length: 5000                             # response 字段
里每个文档的最大字节数。超长截断。不限则设为 0
procs:
    enabled: true
    monitored:
        - process: mysqld
          cmdline_grep: mysqld
        - process: app                                   # 设置发送给 ES
的数据中进程名字段的内容
          cmdline_grep: gunicorn                         # 从 /proc/<pid
>/cmdline 中过滤实际进程名的字符串

output:
    ...
```

在 windows 上，网卡设备名称会比较长。所以 packetbeat 单独提供了一个参数： `packetbeat -device` ，返回整个可用网卡设备列表数组，你可以直接写数组下标来代表这个设备。比如： `device: 0` 。

字段

- `@timestamp` 事件时间
- `type` 事件类型，比如 HTTP, MySQL, Redis, RUM 等
- `count` 事件代表的实际事务数。也就是采样率的倒数
- `direction` 事务流向。比如 "in" 或者 "out"
- `status` 事务状态。不同协议取值方式可能不一致
- `method` 事务方法。对 HTTP 是 GET/POST/PUT 等，对 SQL 是 SELECT/UPDATE/INSERT 等
- `resource` 事务关联的逻辑资源。对 HTTP 是 URL 路径的第一个层级，对 SQL 是表名
- `path` 事务关联的路径。对 HTTP 是 URL 路径，对 SQL 是表名，对 KV 存储是 key
- `query` 人类可读的请求字符串。对 HTTP 是 GET /resource/path?

key=value，对 SQL 是 SELECT id FROM table WHERE name=test
- `params` 请求参数。对 HTTP 就是 GET 或者 POST 的请求参数，对 Thrift 是 request 里的参数
- `notes` packetbeat 解析数据出错时的日志记录

其余根据采用协议不同，各自有自己的字段。

dashboard 效果

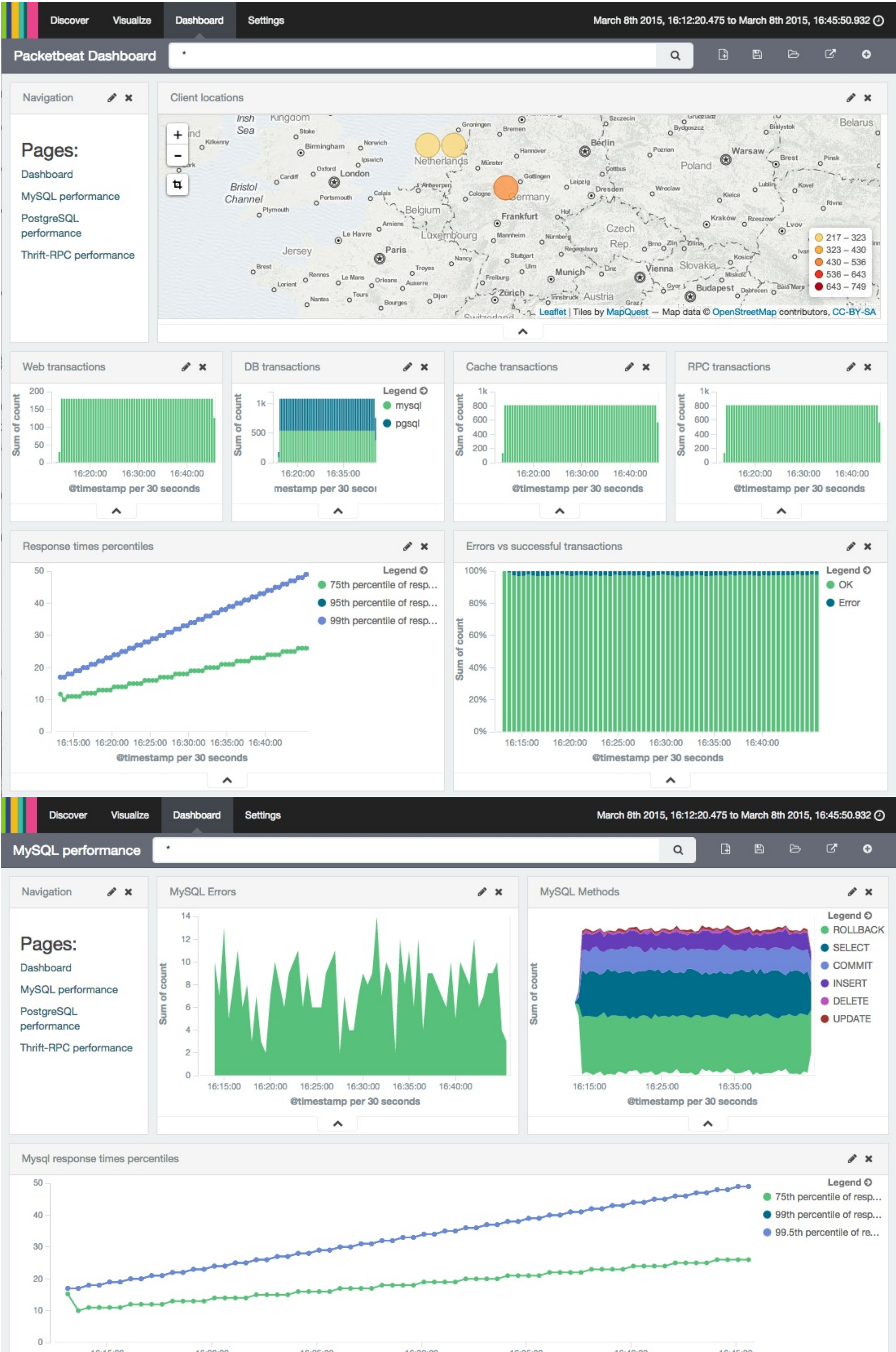
针对 packetbeat 自动识别的不同协议，packetbeat 还自带了几个预定义好的 Kibana dashboard 方便使用和查看。包括：

- Packetbeat Statistics: 针对 HTTP 和标准流量事件的性能统计仪表盘
- Packetbeat Search: 用来搜索关键字的仪表盘
- MySQL Performance: MySQL 性能分析仪表盘
- PgSQL Performance: PgSQL 性能分析仪表盘

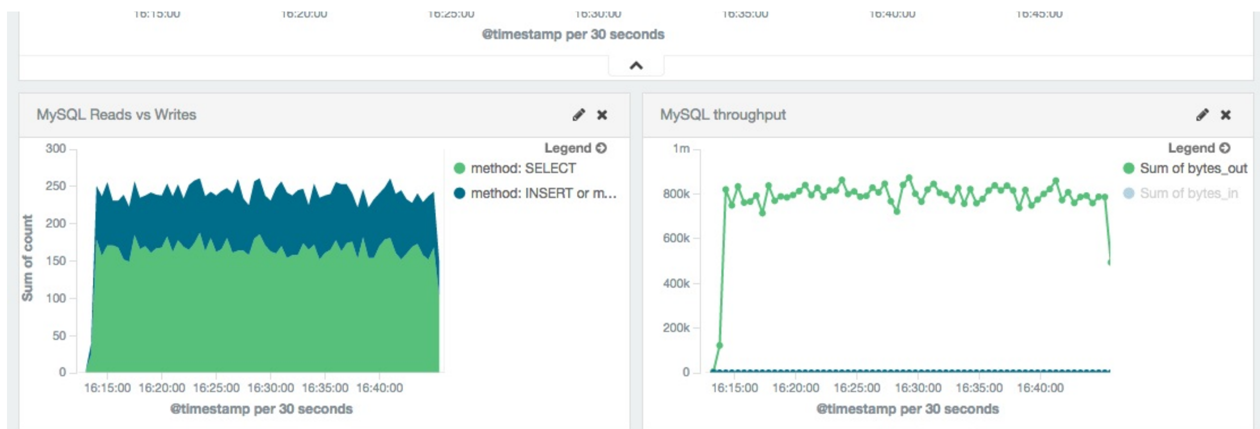
预定义仪表盘的导入方式如下：

```
# git clone https://github.com/elastic/packetbeat-dashboards
# cd packetbeat-dashboards
# ./load.sh http://192.168.0.2:9200
```

效果如下：



Discover
Visualize
Dashboard
Settings
March 8th 2015, 16:12:20.475 to March 8th 2015, 16:45:50.932
Packetbeat Dashboard
Navigation
Pages:
Dashboard
MySQL performance
PostgreSQL performance
Thrift-RPC performance
Client locations
Web transactions
DB transactions
Cache transactions
RPC transactions
@timestamp per 30 seconds
Response times percentiles
Errors vs successful transactions
MySQL performance
MySQL Errors
MySQL Methods
ROLLBACK
SELECT
COMMIT
INSERT
DELETE
UPDATE
Mysql response times percentiles



Kibana3 topology

其实在加入 Elastic.co 之前，packetbeat 曾经自己 fork 了一个 Kibana3 的分支，并在此基础上二次开发了一个专门用来展示网络拓扑结构的面板，叫 force panel。该特性至今依然只能运行在 Kibana3 上。所以，需要网络拓扑展现的用户，还得继续使用 Kibana3。部署方式如下：

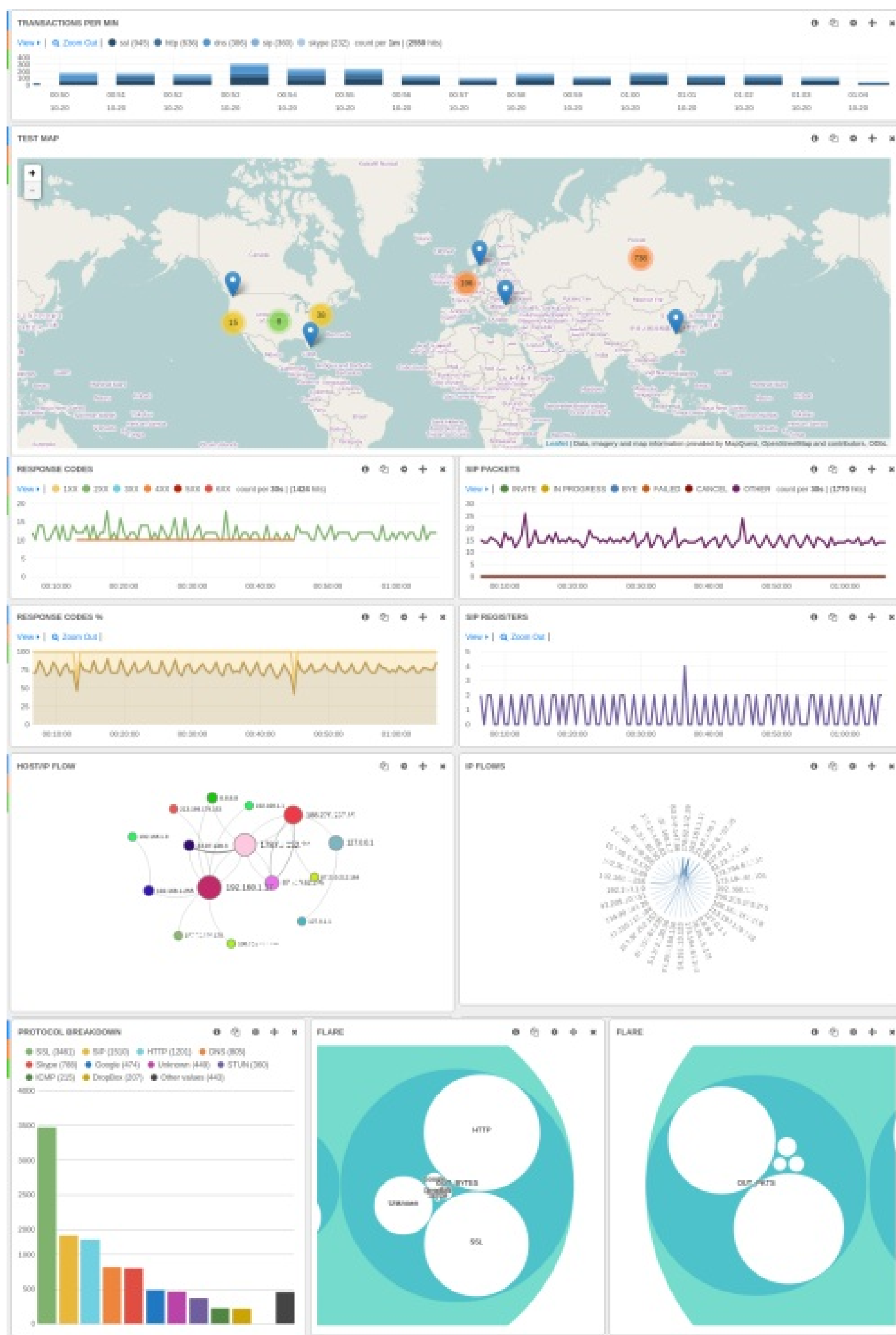
```
curl -L -O https://github.com/packetbeat/kibana/releases/download/v3.1.2-pb/kibana-3.1.2-packetbeat.tar.gz
tar xzvf kibana-3.1.2-packetbeat.tar.gz
curl -L -O https://download.elasticsearch.org/beats/packetbeat/packetbeat-dashboards-k3-1.0.0~Beta1.tar.gz
tar xzvf packetbeat-dashboards-k3-1.0.0~Beta1.tar.gz
cd packetbeat-dashboards-k3-1.0.0~Beta1/
./load.sh 192.169.0.2
```

force panel 示例如下图。注意，force panel 用到的数据，其实质是对各来源 IP 分别请求目的 IP，对 ES 的计算量要求较大，并不适合在高流量高负载的条件下使用。



小贴士

pfring 抓包模式的原厂，ntop 公司，也有类似 packetbeat 的计划。ntopng/nProbe 除了储存到 SQLite 以外，也开始支持存储到 Elasticsearch 中。不过它们推荐采用的 dashboard，是 Kibana3 的另一个 fork 分支，叫 Qbana。



有兴趣的读者可以参考 ntop 官方文档：http://www.ntop.org/ntopng/exploring-your-traffic-using-ntopng-with-elasticsearchkibana/

metricbeat

使用 beat 监控服务性能指标是 ElasticStack 一个常见的使用场景。2.x 时代要求用户对每类常见都需要单独开发自己的 xxxbeat 工具，然后各自编译使用。于是 Elastic.co 公司最终干脆把这件事情统一成了 metricbeat。

目前 metricbeat 支持以下服务性能指标：

- Apache
- HAProxy
- MongoDB
- MySQL
- Nginx
- PostgreSQL
- Redis
- System
- Zookeeper

配置示例

```
metricbeat.modules:
    - module: system
      metricsets:
          - cpu
          - filesystem
          - memory
          - network
          - process
      enabled: true
      period: 10s
      processes: ['.*']
      cpu_ticks: false
    - module: apache
      metricsets: ["status"]
      enabled: true
      period: 1s
      hosts: ["http://127.0.0.1"]
```

Apache

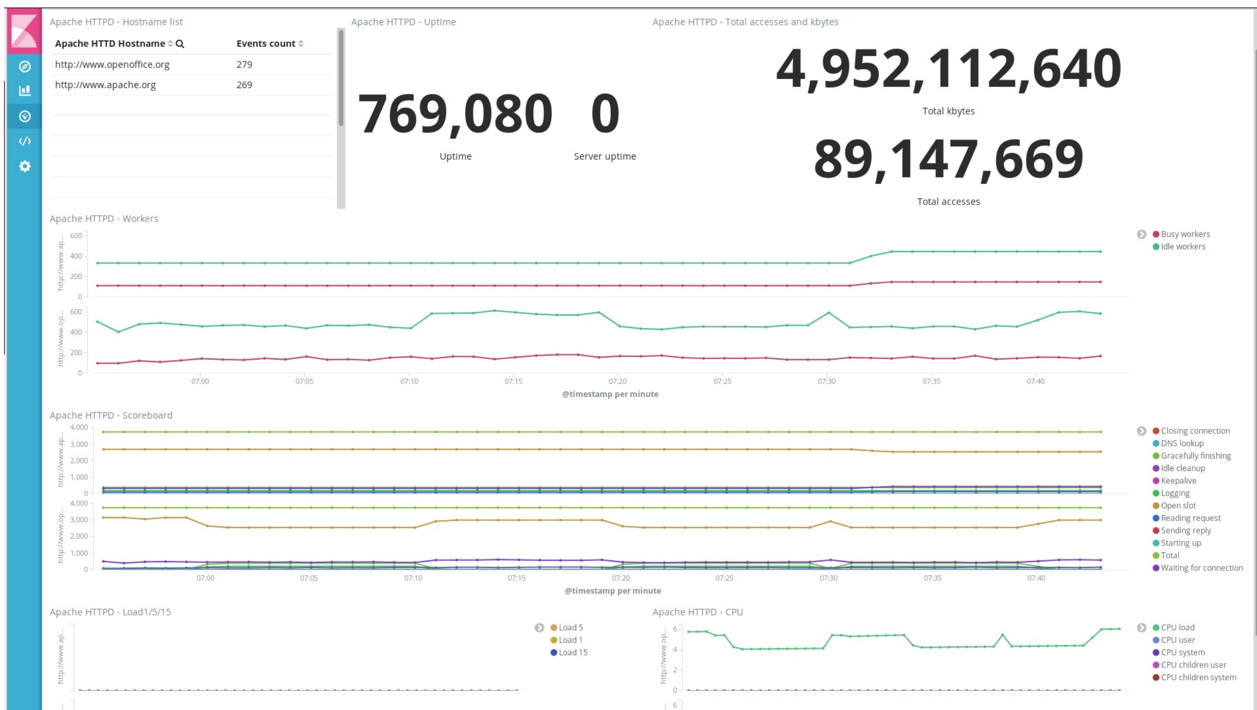
Apache 模块支持 2.2.31 以上的 2.2 系列，或 2.4.16 以上的 2.4 系列版本。

使用该模块要求被监控的 Apache 服务器上安装配置有 `mod_status` 扩展。通过该扩展可以监控到的 status 数据示例如下：

```
    "apache": {
        "status": {
            "bytes_per_request": 1024,
            "bytes_per_sec": 0.201113,
            "connections": {
                "async": {
                    "closing": 0,
                    "keep_alive": 0,
                    "writing": 0
                },
                "total": 0
            },
            "cpu": {
```

```
                "children_system": 0,
                "children_user": 0,
                "load": 0.00652482,
                "system": 1.46,
                "user": 1.53
            },
            "hostname": "apache",
            "load": {
                "1": 0.55,
                "15": 0.31,
                "5": 0.31
            },
            "requests_per_sec": 0.000196399,
            "scoreboard": {
                "closing_connection": 0,
                "dns_lookup": 0,
                "gracefully_finishing": 0,
                "idle_cleanup": 0,
                "keepalive": 0,
                "logging": 0,
                "open_slot": 325,
                "reading_request": 0,
                "sending_reply": 1,
                "starting_up": 0,
                "total": 400,
                "waiting_for_connection": 74
            },
            "total_accesses": 9,
            "total_kbytes": 9,
            "uptime": {
                "server_uptime": 45825,
                "uptime": 45825
            },
            "workers": {
                "busy": 1,
                "idle": 74
            }
        }
    }
```

模块携带有一个预一定好的仪表盘，效果如下：



HAProxy

HAProxy 模块支持 HAProxy 服务器 1.6 版本。

使用该模块要求在 HAProxy 服务器配置的 `global` 或 `default` 区域写有如下配置：

```
stats socket 127.0.0.1:14567
```

模块可以采集两类信息：info 和 stat。

其中 info 的返回数据示例如下：

```
    "haproxy": {
        "info": {
            "compress_bps_in": 0,
            "compress_bps_out": 0,
            "compress_bps_rate_limit": 0,
            "conn_rate": 0,
            "conn_rate_limit": 0,
            "cum_conns": 67,
            "cum_req": 67,
```

```
            "cum_ssl_conns": 0,
            "curr_conns": 0,
            "curr_ssl_conns": 0,
            "hard_max_conn": 4000,
            "idle_pct": 100,
            "max_conn": 4000,
            "max_conn_rate": 5,
            "max_pipes": 0,
            "max_sess_rate": 5,
            "max_sock": 8033,
            "max_ssl_conns": 0,
            "max_ssl_rate": 0,
            "max_zlib_mem_usage": 0,
            "mem_max_mb": 0,
            "nb_proc": 1,
            "pid": 53858,
            "pipes_free": 0,
            "pipes_used": 0,
            "process_num": 1,
            "run_queue": 2,
            "sess_rate": 0,
            "sess_rate_limit": 0,
            "ssl_babckend_key_rate": 0,
            "ssl_backend_max_key_rate": 0,
            "ssl_cache_misses": 0,
            "ssl_cached_lookups": 0,
            "ssl_frontend_key_rate": 0,
            "ssl_frontend_max_key_rate": 0,
            "ssl_frontend_session_reuse_pct": 0,
            "ssl_rate": 0,
            "ssl_rate_limit": 0,
            "tasks": 7,
            "ulimit_n": 8033,
            "uptime_sec": 13700,
            "zlib_mem_usage": 0
        }
    },
```


stat 的返回数据示例如下：

```
    "haproxy": {
        "stat": {
            "act": 1,
            "bck": 0,
            "bin": 0,
            "bout": 0,
            "check_duration": 0,
            "check_status": "L4CON",
            "chkdown": 1,
            "chkfail": 1,
            "cli_abrt": 0,
            "ctime": 0,
            "downtime": 13700,
            "dresp": 0,
            "econ": 0,
            "eresp": 0,
            "hanafail": 0,
            "hrsp_1xx": 0,
            "hrsp_2xx": 0,
            "hrsp_3xx": 0,
            "hrsp_4xx": 0,
            "hrsp_5xx": 0,
            "hrsp_other": 0,
            "iid": 3,
            "last_chk": "Connection refused",
            "lastchg": 13700,
            "lastsess": -1,
            "lbtot": 0,
            "pid": 1,
            "qcur": 0,
            "qmax": 0,
            "qtime": 0,
            "rate": 0,
            "rate_max": 0,
            "rtime": 0,
            "scur": 0,
            "sid": 1,
            "smax": 0,
            "srv_abrt": 0,
            "status": "DOWN",
```

```
                "stot": 0,
                "svname": "log1",
                "ttime": 0,
                "weight": 1,
                "wredis": 0,
                "wretr": 0
            }
    }
```

对这些 stat 数据名称有疑惑的，可以查阅 http://www.haproxy.org/download/1.6/doc/management.txt 文档。

MongoDB

该模块支持 MongoDB 2.8 及以上版本。

```
"mongodb": {
    "status": {
        "asserts": {
            "msg": 0,
            "regular": 0,
            "rollovers": 0,
            "user": 0,
            "warning": 0
        },
        "background_flushing": {
            "average": {
                "ms": 16
            },
            "flushes": 37,
            "last": {
                "ms": 18
            },
            "last_finished": "2016-09-06T07:32:58.228Z",
            "total": {
                "ms": 624
            }
        },
```

```
        "connections": {
            "available": 838859,
            "current": 1,
            "total_created": 10
        },
        "extra_info": {
            "heap_usage": {
                "bytes": 62895448
            },
            "page_faults": 71
        },
        "journaling": {
            "commits": 1,
            "commits_in_write_lock": 0,
            "compression": 0,
            "early_commits": 0,
            "journaled": {
                "mb": 0
            },
            "times": {
                "commits": {
                    "ms": 0
                },
                "commits_in_write_lock": {
                    "ms": 0
                },
                "dt": {
                    "ms": 0
                },
                "prep_log_buffer": {
                    "ms": 0
                },
                "remap_private_view": {
                    "ms": 0
                },
                "write_to_data_files": {
                    "ms": 0
                },
                "write_to_journal": {
                    "ms": 0
```

```
            }
        },
        "write_to_data_files": {
            "mb": 0
        }
    },
    "local_time": "2016-09-06T07:33:15.546Z",
    "memory": {
        "bits": 64,
        "mapped": {
            "mb": 80
        },
        "mapped_with_journal": {
            "mb": 160
        },
        "resident": {
            "mb": 57
        },
        "virtual": {
            "mb": 356
        }
    },
    "network": {
        "in": {
            "bytes": 2258
        },
        "out": {
            "bytes": 93486
        },
        "requests": 39
    },
    "opcounters": {
        "command": 40,
        "delete": 0,
        "getmore": 0,
        "insert": 0,
        "query": 1,
        "update": 0
    },
    "opcounters_replicated": {
```

```
                "command": 0,
                "delete": 0,
                "getmore": 0,
                "insert": 0,
                "query": 0,
                "update": 0
            },
            "storage_engine": {
                "name": "mmapv1"
            },
            "uptime": {
                "ms": 45828938
            },
            "version": "3.0.12",
            "write_backs_queued": false
        }
    }
```


MySQL

该模块支持 MySQL 5.7.0 及以上版本。

```
"mysql": {
    "status": {
        "aborted": {
            "clients": 13,
            "connects": 16
        },
        "binlog": {
            "cache": {
                "disk_use": 0,
                "use": 0
            }
        },
        "bytes": {
            "received": 2100,
            "sent": 92281
        },
        "connections": 33,
        "created": {
            "tmp": {
                "disk_tables": 0,
                "files": 6,
                "tables": 0
            }
        },
        "delayed": {
            "errors": 0,
            "insert_threads": 0,
            "writes": 0
        },
        "flush_commands": 1,
        "max_used_connections": 2,
        "open": {
            "files": 14,
            "streams": 0,
            "tables": 106
        },
        "opened_tables": 113
    }
}
```

Nginx

该模块支持 Nginx 1.9 及以上版本。并要求安装有 `mod_stub_status` 模块。

```
"nginx": {
    "stubstatus": {
        "accepts": 22,
        "active": 1,
        "current": 10,
        "dropped": 0,
        "handled": 22,
        "hostname": "nginx",
        "reading": 0,
        "requests": 10,
        "waiting": 0,
        "writing": 1
    }
}
```

PostgreSQL

该模块支持 PostgreSQL 9 及以上版本。可以采集 activity，bgwriter 和 database 三类数据。

activity 示例数据如下：

```
"postgresql": {
    "activity": {
        "application_name": "",
        "backend_start": "2016-09-06T07:33:18.323Z",
        "client": {
            "address": "172.17.0.14",
            "hostname": "",
            "port": 57436
        },
        "database": {
            "name": "postgres",
            "oid": 12379
        },
        "pid": 162,
        "query": "SELECT * FROM pg_stat_activity",
        "query_start": "2016-09-06T07:33:18.325Z",
        "state": "active",
        "state_change": "2016-09-06T07:33:18.325Z",
        "transaction_start": "2016-09-06T07:33:18.325Z",
        "user": {
            "id": 10,
            "name": "postgres"
        },
        "waiting": false
    },
```


bgwriter 示例数据如下：

```
    "bgwriter": {
        "buffers": {
            "allocated": 191,
            "backend": 0,
            "backend_fsync": 0,
            "checkpoints": 0,
            "clean": 0,
            "clean_full": 0
        },
        "checkpoints": {
            "requested": 0,
            "scheduled": 7,
            "times": {
                "sync": {
                    "ms": 0
                },
                "write": {
                    "ms": 0
                }
            }
        },
        "stats_reset": "2016-09-05T18:49:53.575Z"
    },
```


database 示例数据如下：

```
    "database": {
        "blocks": {
            "hit": 0,
            "read": 0,
            "time": {
                "read": {
                    "ms": 0
                },
                "write": {
                    "ms": 0
                }
            }
        },
        "conflicts": 0,
        "deadlocks": 0,
        "name": "template1",
        "number_of_backends": 0,
        "oid": 1,
        "rows": {
            "deleted": 0,
            "fetched": 0,
            "inserted": 0,
            "returned": 0,
            "updated": 0
        },
        "temporary": {
            "bytes": 0,
            "files": 0
        },
        "transactions": {
            "commit": 0,
            "rollback": 0
        }
    }
```


Redis

该模块支持 Redis 3 及以上版本。可以采集 info 和 keyspace 两类数据。

info 示例数据如下：


```
"redis": {
    "info": {
        "clients": {
            "biggest_input_buf": 0,
            "blocked": 0,
            "connected": 2,
            "longest_output_list": 0
        },
        "cluster": {
            "enabled": false
        },
        "cpu": {
            "used": {
                "sys": 0.33,
                "sys_children": 0,
                "user": 0.39,
                "user_children": 0
            }
        },
        "memory": {
            "allocator": "jemalloc-4.0.3",
            "used": {
                "lua": 37888,
                "peak": 883992,
                "rss": 4030464,
                "value": 883032
            }
        },
        "persistence": {
            "aof": {
                "bgrewrite": {
                    "last_status": "ok"
                },
                "enabled": false,
                "rewrite": {
                    "current_time": {
                        "sec": -1
                    },
```

```
                "in_progress": false,
                "last_time": {
                    "sec": -1
                },
                "scheduled": false
            },
            "write": {
                "last_status": "ok"
            }
        },
        "loading": false,
        "rdb": {
            "bgsave": {
                "current_time": {
                    "sec": -1
                },
                "in_progress": false,
                "last_status": "ok",
                "last_time": {
                    "sec": -1
                }
            },
            "last_save": {
                "changes_since": 2,
                "time": 1475698251
            }
        }
    },
    "replication": {
        "backlog": {
            "active": 0,
            "first_byte_offset": 0,
            "histlen": 0,
            "size": 1048576
        },
        "connected_slaves": 0,
        "master_offset": 0,
        "role": "master"
    },
    "server": {
```

```
            "arch_bits": "64",
            "build_id": "5575d747b4b3b12c",
            "config_file": "",
            "gcc_version": "4.9.2",
            "git_dirty": "0",
            "git_sha1": "00000000",
            "hz": 10,
            "lru_clock": 16080842,
            "mode": "standalone",
            "multiplexing_api": "epoll",
            "os": "Linux 4.4.22-moby x86_64",
            "process_id": 1,
            "run_id": "d37e5ebfe0ae6c4972dbe9f0174a1637bb8247f6"
,
            "tcp_port": 6379,
            "uptime": 383,
            "version": "3.2.4"
        },
        "stats": {
            "commands_processed": 70,
            "connections": {
                "received": 17,
                "rejected": 0
            },
            "instantaneous": {
                "input_kbps": 0.07,
                "ops_per_sec": 2,
                "output_kbps": 0.07
            },
            "keys": {
                "evicted": 0,
                "expired": 0
            },
            "keyspace": {
                "hits": 0,
                "misses": 0
            },
            "latest_fork_usec": 0,
            "migrate_cached_sockets": 0,
            "net": {
```

```
                "input": {
                    "bytes": 1949
                },
                "output": {
                    "bytes": 4956554
                }
            },
            "pubsub": {
                "channels": 0,
                "patterns": 0
            },
            "sync": {
                "full": 0,
                "partial": {
                    "err": 0,
                    "ok": 0
                }
            }
        }
    },
```

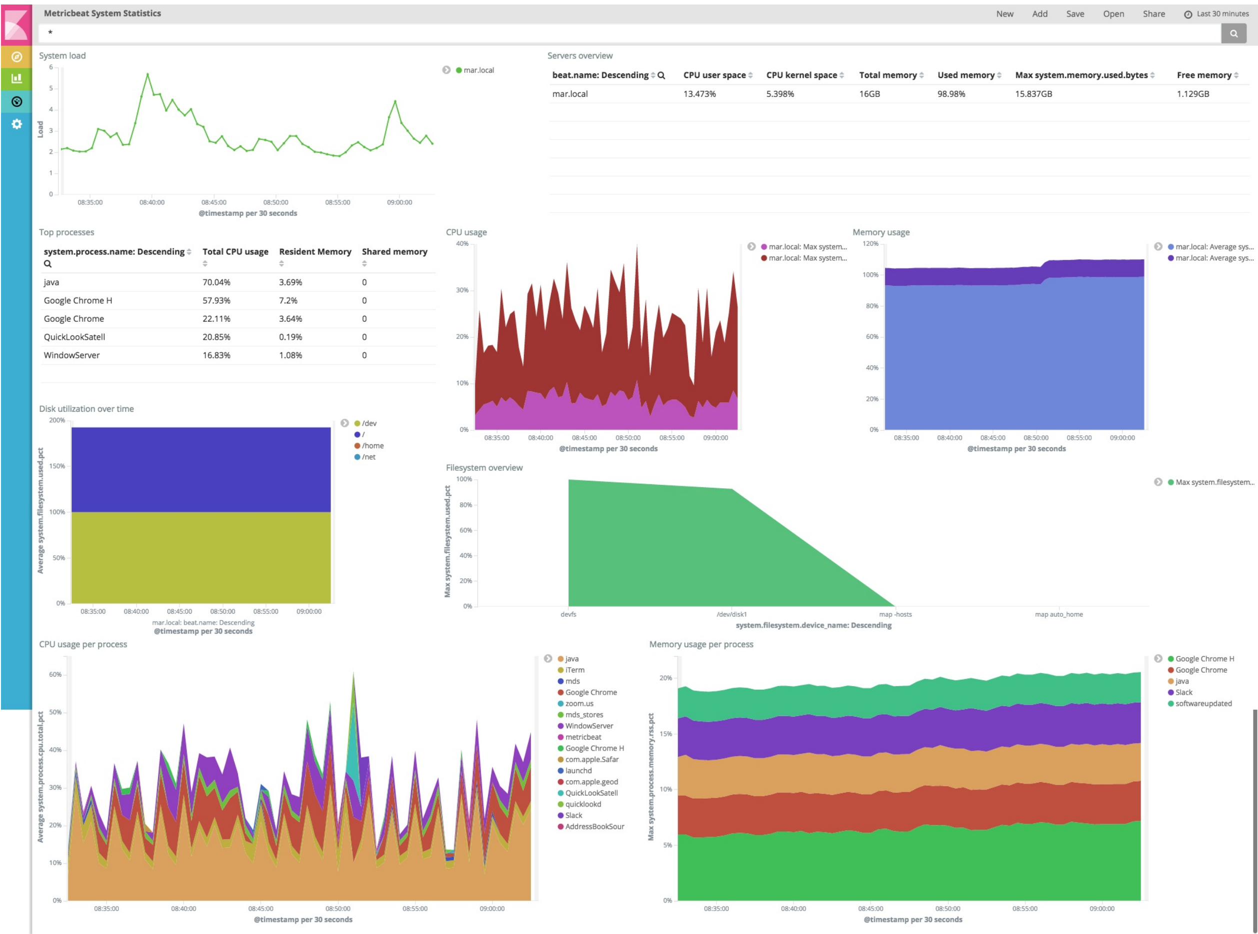
keyspace 示例数据如下：

```
    "keyspace": {
        "avg_ttl": 0,
        "expires": 0,
        "id": "db0",
        "keys": 1
    }
}
```

System

System 就是过去的 TopBeat，可以采集 core、cpu、diskio、filesystem、fsstat、load、memory、network 和 process 指标。这都是运维人员最熟悉的部分，就不再单独贴指标名称和示例了。

模块自带有一个预定义仪表盘，示例如下：



Zookeeper

该模块支持 Zookeeper 3.4.0 及以上版本。采集的 mntr 数据示例如下：

```
"zookeeper": {
    "mntr": {
        "approximate_data_size": 27,
        "ephemerals_count": 0,
        "latency": {
            "avg": 0,
            "max": 0,
            "min": 0
        },
        "num_alive_connections": 1,
        "outstanding_requests": 0,
        "packets": {
            "received": 10,
            "sent": 9
        },
        "server_state": "standalone",
        "version": "3.4.8--1, built on 02/06/2016 03:18 GMT",
        "watch_count": 0,
        "znode_count": 4
    }
}
```

docker 中的采集方式

metricbeat 的 system 数据大多采集自 /proc。而 docker 中，每个容器的实际数据是放在 /hostfs 而不是 /proc 里的。所以如果要用 metricbeat 采集容器数据，需要先挂载好对应路径：

```
$ sudo docker run \
  --volume=/proc:/hostfs/proc:ro \
  --volume=/sys/fs/cgroup:/hostfs/sys/fs/cgroup:ro \
  --volume=/:/hostfs:ro \
  --net=host
  my/metricbeat:latest -system.hostfs=/hostfs
```

winlogbeat

winlogbeat 通过标准的 windows API 获取 windows 系统日志，常见的有 application，hardware，security 和 system 四类。winlogbeat 示例配置如下：

```
winlogbeat.event_logs:
    - name: Application
      provider:
          - Application Error
          - Application Hang
          - Windows Error Reporting
          - EMET
    - name: Security
      level: critical, error, warning
      event_id: 4624, 4625, 4700-4800, -4735
    - name: System
      ignore_older: 168h
    - name: Microsoft-Windows-Windows Defender/Operational
      include_xml: true

output.elasticsearch:
    hosts:
        - localhost:9200
    pipeline: "windows-pipeline-id"

logging.to_files: true
    logging.files:
        path: C:/ProgramData/winlogbeat/Logs
        logging.level: info
```

和其他 beat 一样，这里示例的配置不都是必填项。事实上只有 `event_logs.name` 是必须的。而 winlogbeat 的输出字段中，除了 beats 家族的通用内容外，还包括一下特有字段：

- activity_id
- computer_name：如果运行在 Windows 事件转发模式，这个值会和 beat.hostname 不一样。

- event_data
- event_id
- keywords
- log_name
- level：可选值包括 Success, Information, Warning, Error, Audit Success, and Audit Failure.
- message
- message_error
- record_number
- related_activity_id
- opcode
- provider_guid
- process_id
- source_name
- task
- thread_id
- user_data
- user.identifier
- user.name
- user.domain
- user.type
- version
- xml

Elasticsearch 来源于作者 Shay Banon 的第一个开源项目 Compass 库，而这个 Java 库最初的目的只是为了给 Shay 当时正在学厨师的妻子做一个菜谱的搜索引擎。2010 年，Elasticsearch 正式发布。至今已经成为 GitHub 上最流行的 Java 项目，不过 Shay 承诺给妻子的菜谱搜索依然没有面世……

2015 年初，Elasticsearch 公司召开了第一次全球用户大会 Elastic{ON}15。诸多 IT 巨头纷纷赞助，参会，演讲。会后，Elasticsearch 公司宣布改名 Elastic，公司官网也变成 http://elastic.co/。这意味着 Elasticsearch 的发展方向，不再限于搜索业务，也就是说，Elastic Stack 等机器数据和 IT 服务领域成为官方更加注意的方向。随后几个月，专注监控报警的 Watcher 发布 beta 版，社区有名的网络抓包工具 Packetbeat、多年专注于基于机器学习的异常探测 Prelert 等 ITOA 周边产品纷纷被 Elastic 公司收购。

架构原理

本书作为 Elastic Stack 指南，关注于 Elasticsearch 在日志和数据分析场景的应用，并不打算对底层的 Lucene 原理或者 Java 编程做详细的介绍，但是 Elasticsearch 层面上的一些架构设计，对我们做性能调优，故障处理，具有非常重要的影响。

所以，作为 ES 部分的起始章节，先从数据流向和分布的层面，介绍一下 ES 的工作原理，以及相关的可控项。各位读者可以跳过这节先行阅读后面的运维操作部分，但作为性能调优的基础知识，依然建议大家抽时间返回来了解。

segment、buffer和translog对实时性的影响

既然介绍数据流向，首先第一步就是：写入的数据是如何变成 Elasticsearch 里可以被检索和聚合的索引内容的？

以单文件的静态层面看，每个全文索引都是一个词元的倒排索引，具体涉及到全文索引的通用知识，这里不单独介绍，有兴趣的读者可以阅读《Lucene in Action》等书籍详细了解。

动态更新的 Lucene 索引

以在线动态服务的层面看，要做到实时更新条件下数据的可用和可靠，就需要在倒排索引的基础上，再做一系列更高级的处理。

其实总结一下 Lucene 的处理办法，很简单，就是一句话：新收到的数据写到新的索引文件里。

Lucene 把每次生成的倒排索引，叫做一个段(segment)。然后另外使用一个 commit 文件，记录索引内所有的 segment。而生成 segment 的数据来源，则是内存中的 buffer。也就是说，动态更新过程如下：

1. 当前索引有 3 个 segment 可用。索引状态如图 2-1；

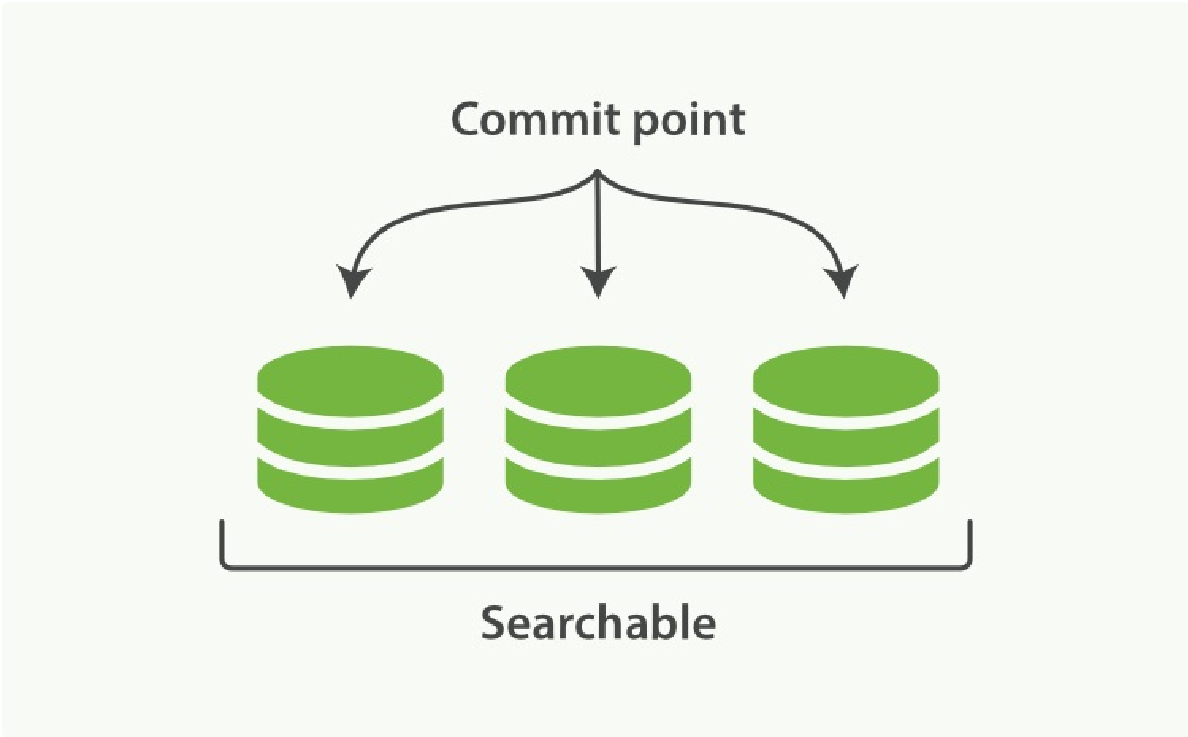


图 2-1

2. 新接收的数据进入内存 buffer。索引状态如图 2-2；

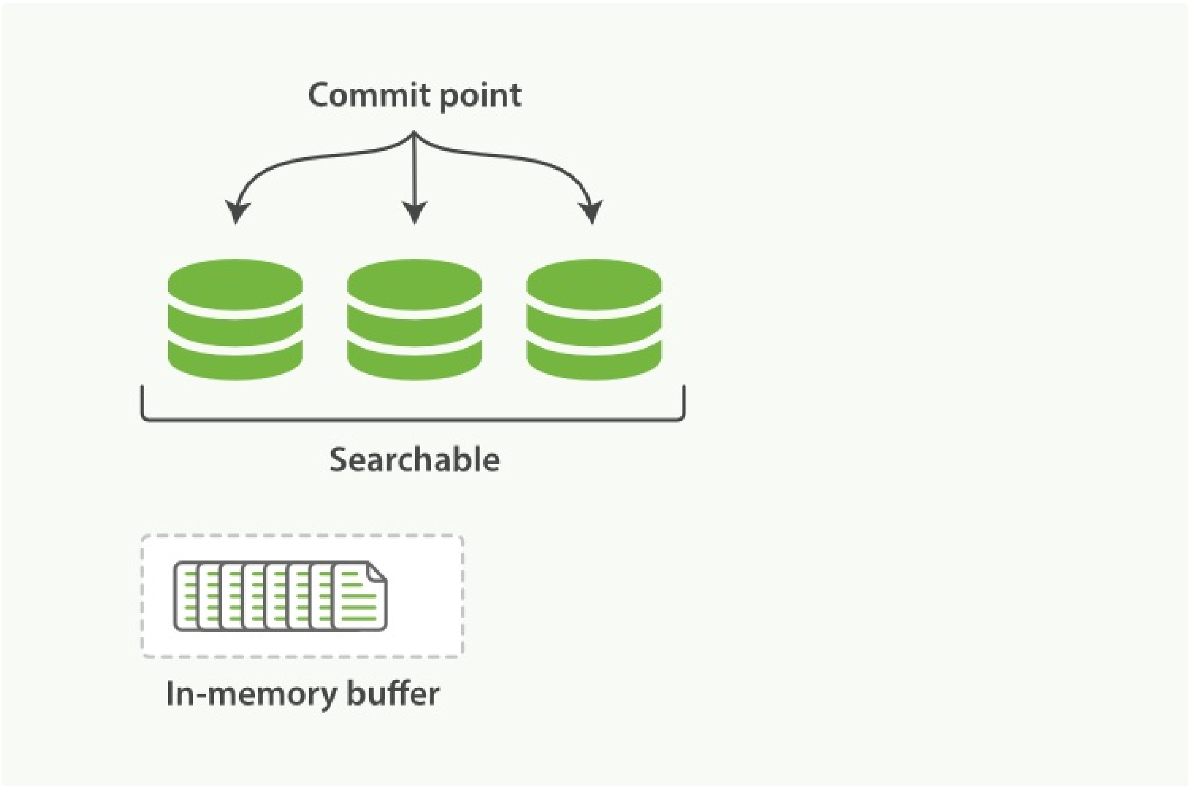


图 2-2

3. 内存 buffer 刷到磁盘，生成一个新的 segment，commit 文件同步更新。索引状态如图 2-3。

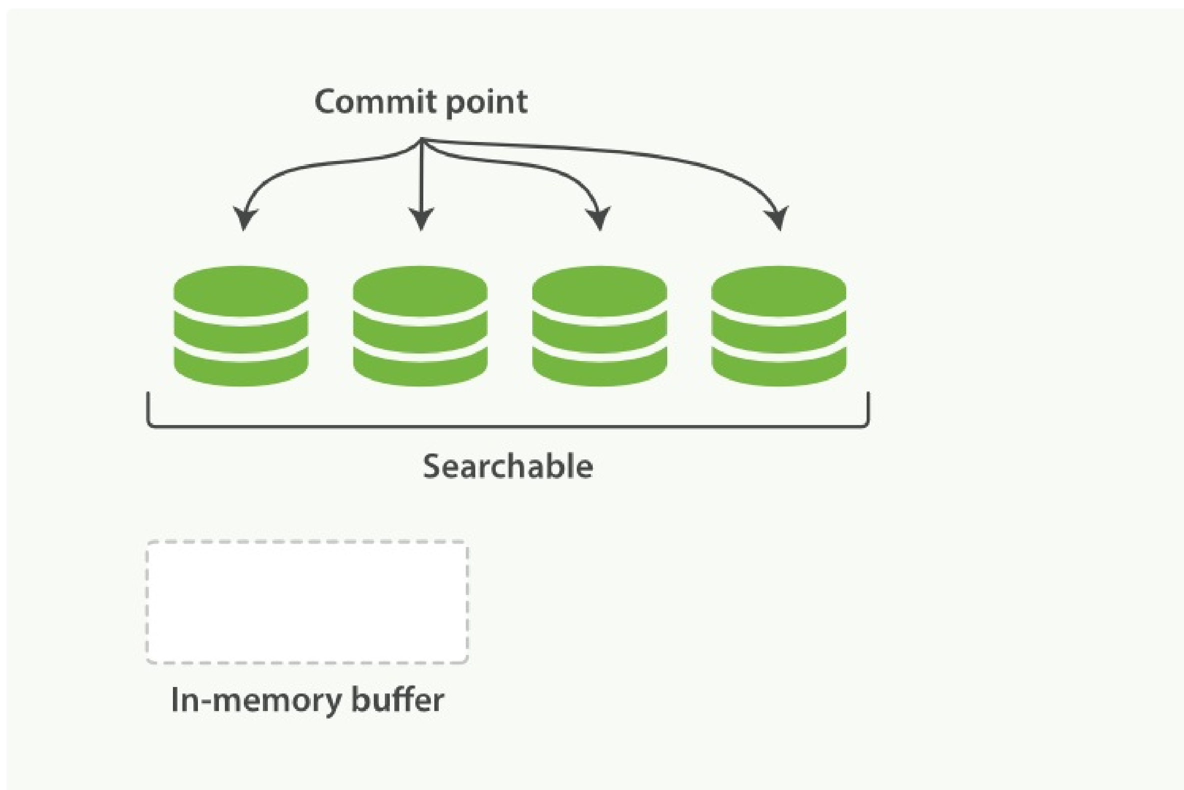


图 2-3

利用磁盘缓存实现的准实时检索

既然涉及到磁盘，那么一个不可避免的问题就来了：磁盘太慢了！对我们要求实时性很高的服务来说，这种处理还不够。所以，在第 3 步的处理中，还有一个中间状态：

1. 内存 buffer 生成一个新的 segment，刷到文件系统缓存中，Lucene 即可检索这个新 segment。索引状态如图 2-4。

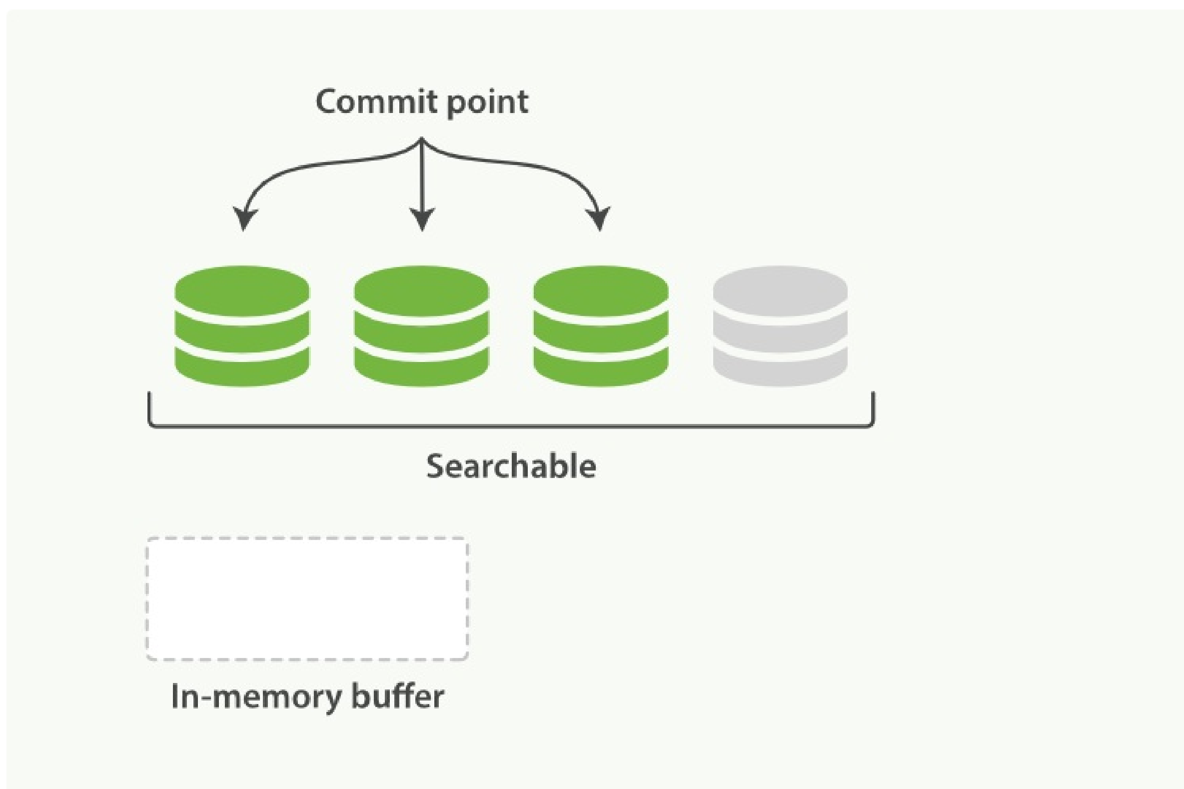


图 2-4

2. 文件系统缓存真正同步到磁盘上，commit 文件更新。达到图 2-3 中的状态。

这一步刷到文件系统缓存的步骤，在 Elasticsearch 中，是默认设置为 1 秒间隔的，对于大多数应用来说，几乎就相当于是实时可搜索了。Elasticsearch 也提供了单独的 `/_refresh` 接口，用户如果对 1 秒间隔还不满意的，可以主动调用该接口来保证搜索可见。

注：*5.0* 中还提供了一个新的请求参数：`?refresh=wait_for`，可以在写入数据后不强制刷新但一直等到刷新才返回。

不过对于 Elastic Stack 的日志场景来说，恰恰相反，我们并不需要如此高的实时性，而是需要更快的写入性能。所以，一般来说，我们反而会通过 `/_settings` 接口或者定制 template 的方式，加大 `refresh_interval` 参数：

```
# curl -XPOST http://127.0.0.1:9200/logstash-2015.06.21/_setting
s -d'
{ "refresh_interval": "10s" }
'
```

如果是导入历史数据的场合，那甚至可以先完全关闭掉：

```
# curl -XPUT http://127.0.0.1:9200/logstash-2015.05.01 -d'
{
  "settings" : {
    "refresh_interval": "-1"
  }
}'
```

在导入完成以后，修改回来或者手动调用一次即可：

```
# curl -XPOST http://127.0.0.1:9200/logstash-2015.05.01/_refresh
```

translog 提供的磁盘同步控制

既然 refresh 只是写到文件系统缓存，那么第 4 步写到实际磁盘又是有什么来控制的？如果这期间发生主机错误、硬件故障等异常情况，数据会不会丢失？

这里，其实有另一个机制来控制。Elasticsearch 在把数据写入到内存 buffer 的同时，其实还另外记录了一个 translog 日志。也就是说，第 2 步并不是图 2-2 的状态，而是像图 2-5 这样：

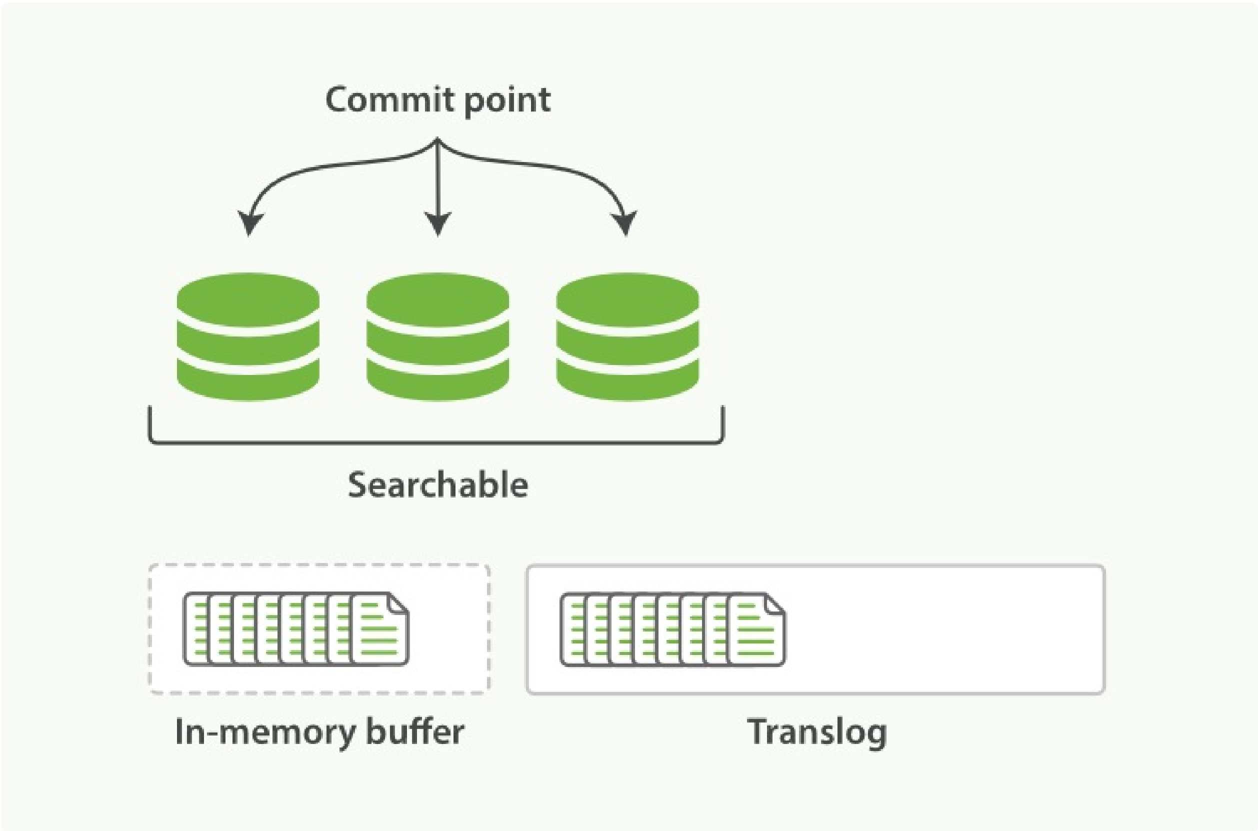


图 2-5

在第 3 和第 4 步，refresh 发生的时候，translog 日志文件依然保持原样，如图 2-6：

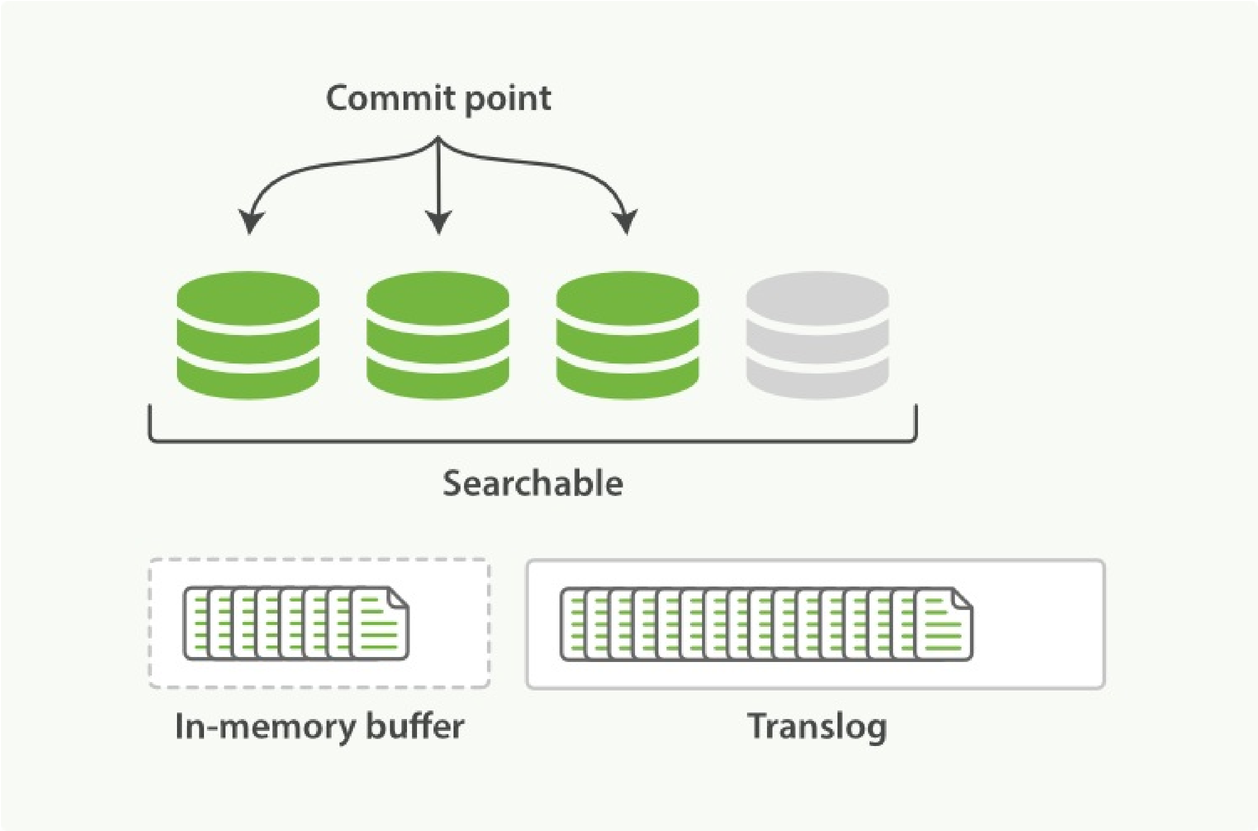


图 2-6

也就是说，如果在这期间发生异常，Elasticsearch 会从 commit 位置开始，恢复整个 translog 文件中的记录，保证数据一致性。

等到真正把 segment 刷到磁盘，且 commit 文件进行更新的时候， translog 文件才清空。这一步，叫做 flush。同样，Elasticsearch 也提供了 `/_flush` 接口。

对于 flush 操作，Elasticsearch 默认设置为：每 30 分钟主动进行一次 flush，或者当 translog 文件大小大于 512MB (老版本是 200MB)时，主动进行一次 flush。这两个行为，可以分别通过 `index.translog.flush_threshold_period` 和 `index.translog.flush_threshold_size` 参数修改。

如果对这两种控制方式都不满意，Elasticsearch 还可以通过 `index.translog.flush_threshold_ops` 参数，控制每收到多少条数据后 flush 一次。

translog 的一致性

索引数据的一致性通过 translog 保证。那么 translog 文件自己呢？

默认情况下，Elasticsearch 每 5 秒，或每次请求操作结束前，会强制刷新 translog 日志到磁盘上。

后者是 Elasticsearch 2.0 新加入的特性。为了保证不丢数据，每次 index、bulk、delete、update 完成的时候，一定触发刷新 translog 到磁盘上，才给请求返回 200 OK。这个改变在提高数据安全性的同时当然也降低了一点性能。

如果你不在意这点可能性，还是希望性能优先，可以在 index template 里设置如下参数：

```
{
    "index.translog.durability": "async"
}
```

Elasticsearch 分布式索引

大家可能注意到了，前面一段内容，一直写的是"Lucene 索引"。这个区别在于，Elasticsearch 为了完成分布式系统，对一些名词概念作了变动。索引成为了整个集群级别的命名，而在单个主机上的*Lucene* 索引，则被命名为分片 *(shard)*。至于数

据是怎么识别到自己应该在哪个分片，请阅读稍后有关 routing 的章节。

segment merge对写入性能的影响

通过上节内容，我们知道了数据怎么进入 ES 并且如何才能让数据更快的被检索使用。其中用一句话概括了 Lucene 的设计思路就是"开新文件"。从另一个方面看，开新文件也会给服务器带来负载压力。因为默认每 1 秒，都会有一个新文件产生，每个文件都需要有文件句柄，内存，CPU 使用等各种资源。一天有 86400 秒，设想一下，每次请求要扫描一遍 86400 个文件，这个响应性能绝对好不了！

为了解决这个问题，ES 会不断在后台运行任务，主动将这些零散的 segment 做数据归并，尽量让索引内只保有少量的，每个都比较大的，segment 文件。这个过程是有独立的线程来进行的，并不影响新 segment 的产生。归并过程中，索引状态如图 2-7，尚未完成的较大的 segment 是被排除在检索可见范围之外的：

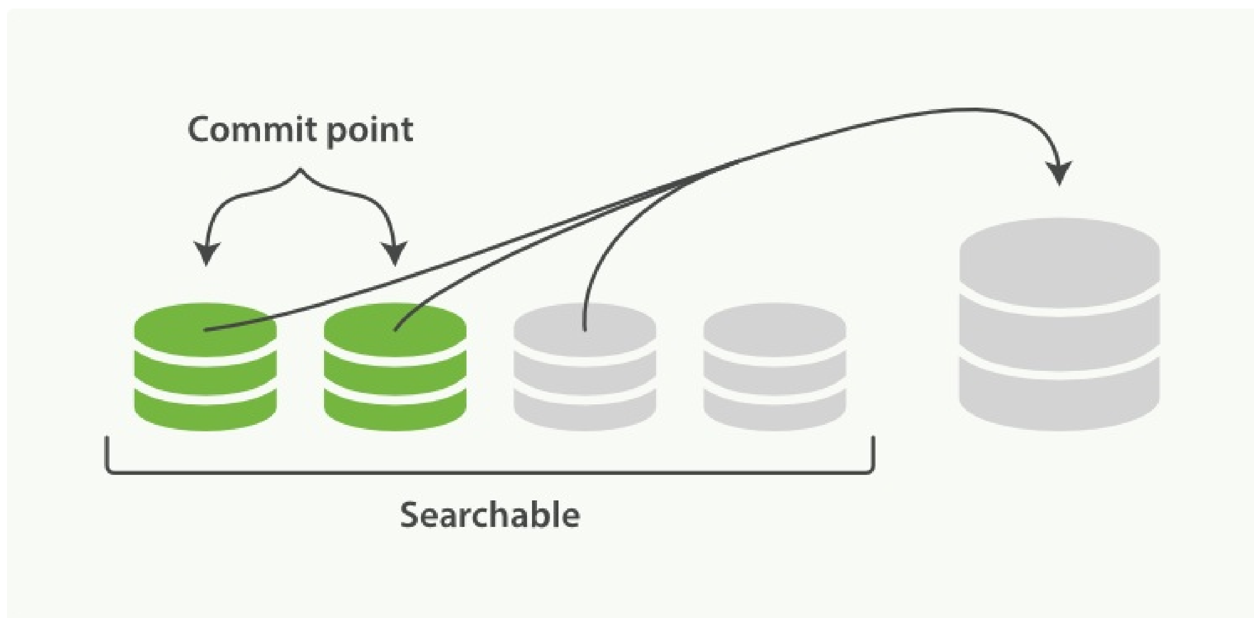


图 2-7

当归并完成，较大的这个 segment 刷到磁盘后，commit 文件做出相应变更，删除之前几个小 segment，改成新的大 segment。等检索请求都从小 segment 转到大 segment 上以后，删除没用的小 segment。这时候，索引里 segment 数量就下降了，状态如图 2-8 所示：

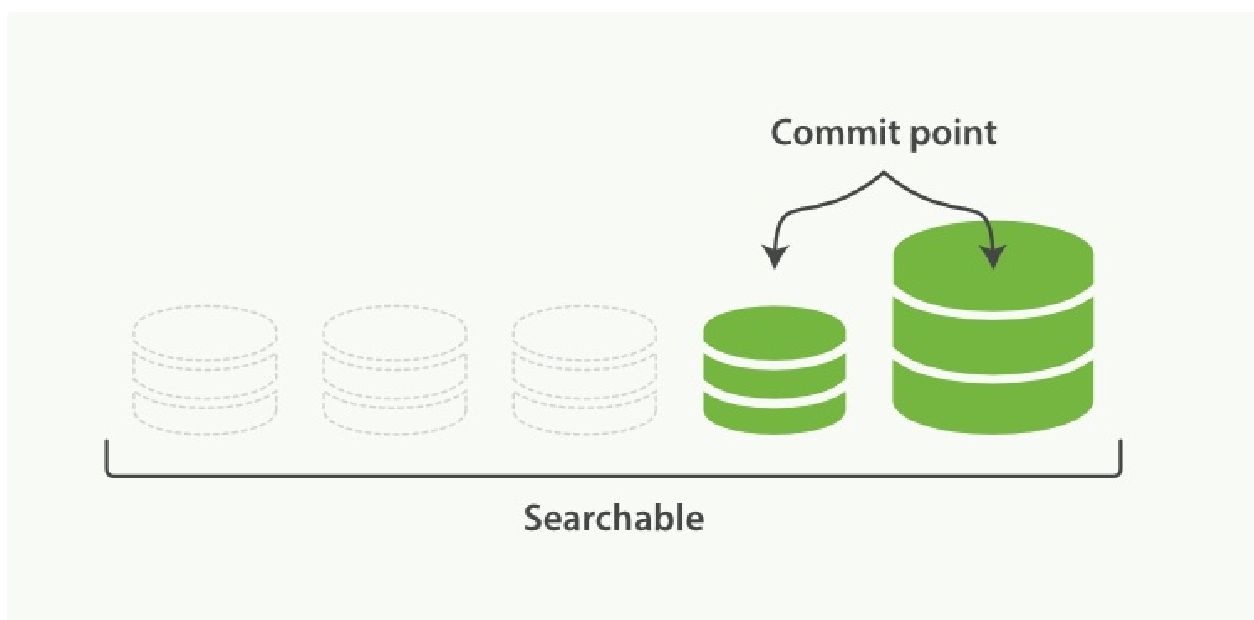


图 2-8

归并线程配置

segment 归并的过程，需要先读取 segment，归并计算，再写一遍 segment，最后还要保证刷到磁盘。可以说，这是一个非常消耗磁盘 IO 和 CPU 的任务。所以，ES 提供了对归并线程的限速机制，确保这个任务不会过分影响到其他任务。

在 5.0 之前，归并线程的限速配置 `indices.store.throttle.max_bytes_per_sec` 是 20MB。对于写入量较大，磁盘转速较高，甚至使用 SSD 盘的服务器来说，这个限速是明显过低的。对于 Elastic Stack 应用，社区广泛的建议是可以适当调大到 100MB或者更高。

```
# curl -XPUT http://127.0.0.1:9200/_cluster/settings -d'
{
    "persistent" : {
        "indices.store.throttle.max_bytes_per_sec" : "100mb"
    }
}'
```

5.0 开始，ES 对此作了大幅度改进，使用了 Lucene 的 CMS(ConcurrentMergeScheduler) 的 auto throttle 机制，正常情况下已经不再需要手动配置 `indices.store.throttle.max_bytes_per_sec` 了。官方文档中都已经删除了相关介绍，不过从源码中还是可以看到，这个值目前的默认设置是 10240 MB。

归并线程的数目，ES 也是有所控制的。默认数目的计算公式是： `Math.min(3, Runtime.getRuntime().availableProcessors() / 2)` 。即服务器 CPU 核数的一半大于 3 时，启动 3 个归并线程；否则启动跟 CPU 核数的一半相等的线程数。相信一般做 Elastic Stack 的服务器 CPU 合数都会在 6 个以上。所以一般来说就是 3 个归并线程。如果你确定自己磁盘性能跟不上，可以降低 `index.merge.scheduler.max_thread_count` 配置，免得 IO 情况更加恶化。

归并策略

归并线程是按照一定的运行策略来挑选 segment 进行归并的。主要有以下几条：

- index.merge.policy.floor_segment 默认 2MB，小于这个大小的 segment，优先被归并。
- index.merge.policy.max_merge_at_once 默认一次最多归并 10 个 segment
- index.merge.policy.max_merge_at_once_explicit 默认 forcemerge 时一次最多归并 30 个 segment。
- index.merge.policy.max_merged_segment 默认 5 GB，大于这个大小的 segment，不用参与归并。forcemerge 除外。

根据这段策略，其实我们也可以从另一个角度考虑如何减少 segment 归并的消耗以及提高响应的办法：加大 flush 间隔，尽量让每次新生成的 segment 本身大小就比较大。

forcemerge 接口

既然默认的最大 segment 大小是 5GB。那么一个比较庞大的数据索引，就必然会有为数不少的 segment 永远存在，这对文件句柄，内存等资源都是极大的浪费。但是由于归并任务太消耗资源，所以一般不太选择加大 `index.merge.policy.max_merged_segment` 配置，而是在负载较低的时间段，通过 forcemerge 接口，强制归并 segment。

```
# curl -XPOST http://127.0.0.1:9200/logstash-2015-06.10/_forceme
rge?max_num_segments=1
```

由于 forcemerge 线程对资源的消耗比普通的归并线程大得多，所以，绝对不建议对还在写入数据的热索引执行这个操作。这个问题对于 Elastic Stack 来说非常好办，一般索引都是按天分割的。更合适的任务定义方式，请阅读本书稍后的 curator 章节。

routing和replica的读写过程

之前两节，完整介绍了在单个 Lucene 索引，即 ES 分片内的数据写入流程。现在彻底回到 ES 的分布式层面上来，当一个 ES 节点收到一条数据的写入请求时，它是如何确认这个数据应该存储在哪个节点的哪个分片上的？

路由计算

作为一个没有额外依赖的简单的分布式方案，ES 在这个问题上同样选择了一个非常简洁的处理方式，对任一条数据计算其对应分片的方式如下：

> shard = hash(routing) % number_of_primary_shards

每个数据都有一个 routing 参数，默认情况下，就使用其 `_id` 值。将其 `_id` 值计算哈希后，对索引的主分片数取余，就是数据实际应该存储到的分片 ID。

由于取余这个计算，完全依赖于分母，所以导致 ES 索引有一个限制，索引的主分片数，不可以随意修改。因为一旦主分片数不一样，所以数据的存储位置计算结果都会发生改变，索引数据就完全不可读了。

副本一致性

作为分布式系统，数据副本可算是一个标配。ES 数据写入流程，自然也涉及到副本。在有副本配置的情况下，数据从发向 ES 节点，到接到 ES 节点响应返回，流向如下(附图 2-9)：

1. 客户端请求发送给 Node 1 节点，注意图中 Node 1 是 Master 节点，实际完全可以不是。
2. Node 1 用数据的 `_id` 取余计算得到应该讲数据存储到 shard 0 上。通过 cluster state 信息发现 shard 0 的主分片已经分配到了 Node 3 上。Node 1 转发请求数据给 Node 3。
3. Node 3 完成请求数据的索引过程，存入主分片 0。然后并行转发数据给分配有 shard 0 的副本分片的 Node 1 和 Node 2。当收到任一节点汇报副本分片数据写入成功，Node 3 即返回给初始的接收节点 Node 1，宣布数据写入成功。Node 1 返回成功响应给客户端。

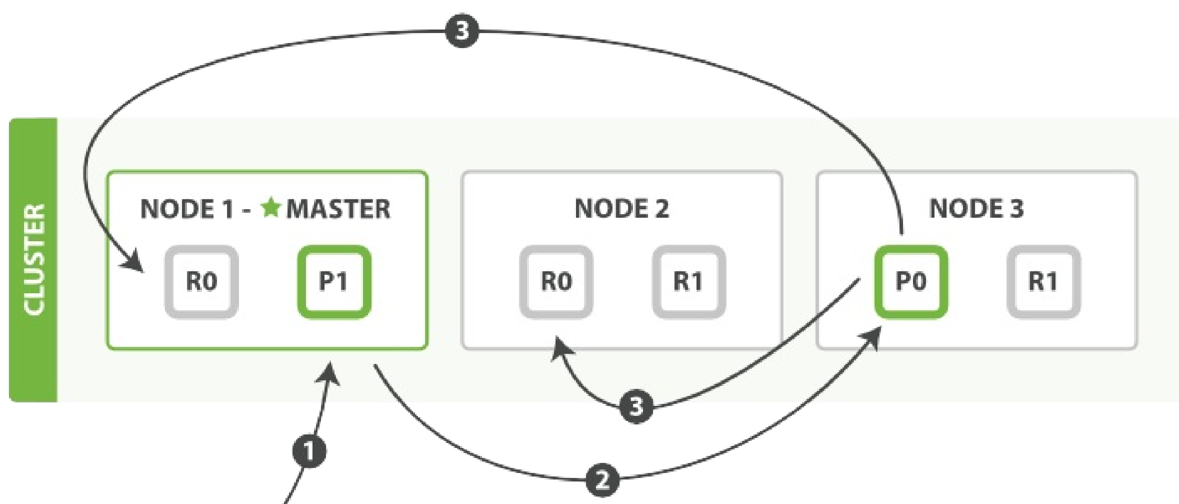


图 2-9

这个过程中，有几个参数可以用来控制或变更其行为：

- wait_for_active_shards 上面示例中，2 个副本分片只要有 1 个成功，就可以返回给客户端了。这点也是有配置项的。其默认值的计算来源如下：

> int((primary + number_of_replicas) / 2) + 1

根据需要，也可以将参数设置为 one，表示仅写完主分片就返回，等同于 async；还可以设置为 all，表示等所有副本分片都写完才能返回。

- timeout 如果集群出现异常，有些分片当前不可用，ES 默认会等待 1 分钟看分片能否恢复。可以使用 `?timeout=30s` 参数来缩短这个等待时间。

副本配置和分片配置不一样，是可以随时调整的。有些较大的索引，甚至可以在做 forcemerge 前，先把副本全部取消掉，等 optimize 完后，再重新开启副本，节约单个 segment 的重复归并消耗。

```
# curl -XPUT http://127.0.0.1:9200/logstash-mweibo-2015.05.02/_s
ettings -d '{
    "index": { "number_of_replicas" : 0 }
}'
```

shard 的 allocate 控制

某个 shard 分配在哪个节点上，一般来说，是由 ES 自动决定的。以下几种情况会触发分配动作：

1. 新索引生成
2. 索引的删除
3. 新增副本分片
4. 节点增减引发的数据均衡

ES 提供了一系列参数详细控制这部分逻辑：

- cluster.routing.allocation.enable 该参数用来控制允许分配哪种分片。默认是 `all` 。可选项还包括 `primaries` 和 `new_primaries` 。 `none` 则彻底拒绝分片。该参数的作用，本书稍后集群升级章节会有说明。
- cluster.routing.allocation.allow_rebalance 该参数用来控制什么时候允许数据均衡。默认是 `indices_all_active` ，即要求所有分片都正常启动成功以后，才可以进行数据均衡操作，否则的话，在集群重启阶段，会浪费太多流量了。
- cluster.routing.allocation.cluster_concurrent_rebalance 该参数用来控制集群内同时运行的数据均衡任务个数。默认是 2 个。如果有节点增减，且集群负载压力不高的时候，可以适当加大。
- cluster.routing.allocation.node_initial_primaries_recoveries 该参数用来控制节点重启时，允许同时恢复几个主分片。默认是 4 个。如果节点是多磁盘，且 IO 压力不大，可以适当加大。
- cluster.routing.allocation.node_concurrent_recoveries 该参数用来控制节点除了主分片重启恢复以外其他情况下，允许同时运行的数据恢复任务。默认是 2 个。所以，节点重启时，可以看到主分片迅速恢复完成，副本分片的恢复却很慢。除了副本分片本身数据要通过网络复制以外，并发线程本身也减少了一半。当然，这种设置也是有道理的——主分片一定是本地恢复，副本分片却需要走网络，带宽是有限的。从 ES 1.6 开始，冷索引的副本分片可以本地恢复，这个参数也就是可以适当加大了。
- indices.recovery.concurrent_streams 该参数用来控制节点从网络复制恢复副本分片时的数据流个数。默认是 3 个。可以配合上一条配置一起加大。
- indices.recovery.max_bytes_per_sec 该参数用来控制节点恢复时的速率。默认是 40MB。显然是比较小的，建议加大。

此外，ES 还有一些其他的分片分配控制策略。比如以 `tag` 和 `rack_id` 作为区分等。一般来说，Elastic Stack 场景中使用不多。运维人员可能比较常见的策略有两种：

1. 磁盘限额 为了保护节点数据安全，ES 会定时(`cluster.info.update.interval` ，默认 30 秒)检查一下各节点的数据目录磁盘使用情况。在达到 `cluster.routing.allocation.disk.watermark.low` (默认 85%)的时候，新索引分片就不会再分配到这个节点上了。在达到 `cluster.routing.allocation.disk.watermark.high` (默认 90%)的时候，就会触发该节点现存分片的数据均衡，把数据挪到其他节点上去。这两个值不但可以写百分比，还可以写具体的字节数。有些公司可能出于成本考虑，对磁盘使用率有一定的要求，需要适当抬高这个配置：

```
# curl -XPUT localhost:9200/_cluster/settings -d '{
    "transient" : {
        "cluster.routing.allocation.disk.watermark.low" : "85%",
        "cluster.routing.allocation.disk.watermark.high" : "10gb
",
        "cluster.info.update.interval" : "1m"
    }
}'
```

1. 热索引分片不均 默认情况下，ES 集群的数据均衡策略是以各节点的分片总数(*indices_all_active*)作为基准的。这对于搜索服务来说无疑是均衡搜索压力提高性能的好办法。但是对于 Elastic Stack 场景，一般压力集中在新索引的数据写入方面。正常运行的时候，也没有问题。但是当集群扩容时，新加入集群的节点，分片总数远远低于其他节点。这时候如果有新索引创建，ES 的默认策略会导致新索引的所有主分片几乎全分配在这台新节点上。整个集群的写入压力，压在一个节点上，结果很可能是这个节点直接被压死，集群出现异常。 所以，对于 Elastic Stack 场景，强烈建议大家预先计算好索引的分片数后，配置好单节点分片的限额。比如，一个 5 节点的集群，索引主分片 10 个，副本 1 份。则平均下来每个节点应该有 4 个分片，那么就配置：

```
# curl -s -XPUT http://127.0.0.1:9200/logstash-2015.05.08/_setti
ngs -d '{
    "index": { "routing.allocation.total_shards_per_node" : "5"
}
}'
```

注意，这里配置的是 5 而不是 4。因为我们需要预防有机器故障，分片发生迁移的情况。如果写的是 4，那么分片迁移会失败。

此外，另一种方式则更加玄妙，Elasticsearch 中有一系列参数，相互影响，最终联合决定分片分配：

- cluster.routing.allocation.balance.shard 节点上分配分片的权重，默认为 0.45。数值越大越倾向于在节点层面均衡分片。
- cluster.routing.allocation.balance.index 每个索引往单个节点上分配分片的权重，默认为 0.55。数值越大越倾向于在索引层面均衡分片。
- cluster.routing.allocation.balance.threshold 大于阈值则触发均衡操作。默认为 1。

Elasticsearch 中的计算方法是：

(indexBalance *(node.numShards(index) – avgShardsPerNode(index))* + *shardBalance* (node.numShards() – avgShardsPerNode)) <=> weightthreshold

所以，也可以采取加大 `cluster.routing.allocation.balance.index` ，甚至设置 `cluster.routing.allocation.balance.shard` 为 0 来尽量采用索引内的节点均衡。

reroute 接口

上面说的各种配置，都是从策略层面，控制分片分配的选择。在必要的时候，还可以通过 ES 的 reroute 接口，手动完成对分片的分配选择的控制。

reroute 接口支持五种指令： `allocate_replica` , `allocate_stale_primary` , `allocate_empty_primary` ， `move` 和 `cancel` 。常用的一般是 allocate 和 move：

- `allocate_*` 指令

因为负载过高等原因，有时候个别分片可能长期处于 UNASSIGNED 状态，我们就可以手动分配分片到指定节点上。默认情况下只允许手动分配副本分片(即使用 `allocate_replica`)，所以如果要分配主分片，需要单独加一个 `accept_data_loss` 选项：

```
# curl -XPOST 127.0.0.1:9200/_cluster/reroute -d '{
  "commands" : [ {
        "allocate_stale_primary" :
            {
              "index" : "logstash-2015.05.27", "shard" : 61, "no
de" : "10.19.0.77", "accept_data_loss" : true
            }
        }
  ]
}'
```

注意， `allocate_stale_primary` 表示准备分配到的节点上可能有老版本的历史数据，运行时请提前确认一下是哪个节点上保留有这个分片的实际目录，且目录大小最大。然后手动分配到这个节点上。以此减少数据丢失。

- move 指令

因为负载过高，磁盘利用率过高，服务器下线，更换磁盘等原因，可以会需要从节点上移走部分分片：

```
curl -XPOST 127.0.0.1:9200/_cluster/reroute -d '{
  "commands" : [ {
        "move" :
            {
              "index" : "logstash-2015.05.22", "shard" : 0, "fro
m_node" : "10.19.0.81", "to_node" : "10.19.0.104"
            }
        }
  ]
}'
```

分配失败原因

如果是自己手工 reroute 失败，Elasticsearch 返回的响应中会带上失败的原因。不过格式非常难看，一堆 YES，NO。从 5.0 版本开始，Elasticsearch 新增了一个 allocation explain 接口，专门用来解释指定分片的具体失败理由：

`` curl -XGET 'http://localhost:9200/_cluster/allocation/explain' -d'{ "index": "logstash-2016.10.31", "shard": 0, "primary": false

}'

```
得到的响应如下：
```

{ "shard" : { "index" : "myindex", "index_uuid" : "KnW0-zELRs6PK84l0r38ZA", "id" : 0, "primary" : false }, "assigned" : false, "shard_state_fetch_pending": false, "unassigned_info" : { "reason" : "INDEX_CREATED", "at" : "2016-03-22T20:04:23.620Z" }, "allocation_delay_ms" : 0, "remaining_delay_ms" : 0, "nodes" : { "V-Spi0AyRZ6ZvKbaI3691w" : { "node_name" : "H5dfFeA", "node_attributes" : { "bar" : "baz" }, "store" : { "shard_copy" : "NONE" }, "final_decision" : "NO", "final_explanation" : "the shard cannot be assigned because one or more allocation decider returns a 'NO' decision", "weight" : 0.06666675, "decisions" : [{ "decider" : "filter", "decision" : "NO", "explanation" : "node does not match index include filters [foo:\"bar\"]" }] }, "Qc6VL8c5RWaw1qXZ0Rg57g" : { ...

```
这会是很长一串 JSON，把集群里所有的节点都列上来，挨个解释为什么不能分配到这个节点。

## 节点下线

集群中个别节点出现故障预警等情况，需要下线，也是 Elasticsearch 运维工作中常见的情况。如果已经稳定运行过一段时间的集群，每个节点上都会保存有数量不少的分片。这种时候通过 reroute 接口手动转移，就显得太过麻烦了。这个时候，有另一种方式：
```

curl -XPUT 127.0.0.1:9200/_cluster/settings -d '{ "transient" :{ "cluster.routing.allocation.exclude._ip" : "10.0.0.1" } }'

```
Elasticsearch 集群就会自动把这个 IP 上的所有分片，都自动转移到其他节点上
。等到转移完成，这个空节点就可以毫无影响的下线了。

和 `_ip` 类似的参数还有 `_host`, `_name` 等。此外，这类参数不单是 clust
er 级别，也可以是 index 级别。下一小节就是 index 级别的用例。

## 冷热数据的读写分离

Elasticsearch 集群一个比较突出的问题是: 用户做一次大的查询的时候, 非常大
量的读 IO 以及聚合计算导致机器 Load 升高, CPU 使用率上升, 会影响阻塞到新
数据的写入, 这个过程甚至会持续几分钟。所以，可能需要仿照 MySQL 集群一样，做
读写分离。

### 实施方案

1. N 台机器做热数据的存储, 上面只放当天的数据。这 N 台热数据节点上面的 ela
sticsearc.yml 中配置 `node.attr.tag: hot`
2. 之前的数据放在另外的 M 台机器上。这 M 台冷数据节点中配置 `node.attr.t
ag: stale`
3. 模板中控制对新建索引添加 hot 标签：
```

{ "order" : 0, "template" : "*", "settings" : { "index.routing.allocation.include.tag" : "hot" } }

```
4. 每天计划任务更新索引的配置, 将 tag 更改为 stale, 索引会自动迁移到 M 台
冷数据节点
```

curl -XPUT http://127.0.0.1:9200/indexname/_settings -d'

{ "index": { "routing": { "allocation": { "include": { "tag": "stale" } } } } }' ```

这样，写操作集中在 N 台热数据节点上，大范围的读操作集中在 M 台冷数据节点上。避免了堵塞影响。

该方案运用的，是 Elasticsearch 中的 allocation filter 功能，详细说明见：https://www.elastic.co/guide/en/elasticsearch/reference/master/shard-allocation-filtering.html

集群自动发现

ES 是一个 P2P 类型(使用 gossip 协议)的分布式系统，除了集群状态管理以外，其他所有的请求都可以发送到集群内任意一台节点上，这个节点可以自己找到需要转发给哪些节点，并且直接跟这些节点通信。

所以，从网络架构及服务配置上来说，构建集群所需要的配置极其简单。在 Elasticsearch 2.0 之前，无阻碍的网络下，所有配置了相同 `cluster.name` 的节点都自动归属到一个集群中。

2.0 版本之后，基于安全的考虑，Elasticsearch 稍作了调整，避免开发环境过于随便造成的麻烦。

unicast 方式

ES 从 2.0 版本开始，默认的自动发现方式改为了单播(unicast)方式。配置里提供几台节点的地址，ES 将其视作 gossip router 角色，借以完成集群的发现。由于这只是 ES 内一个很小的功能，所以 gossip router 角色并不需要单独配置，每个 ES 节点都可以担任。所以，采用单播方式的集群，各节点都配置相同的几个节点列表作为 router 即可。

此外，考虑到节点有时候因为高负载，慢 GC 等原因可能会有偶尔没及时响应 ping 包的可能，一般建议稍微加大 Fault Detection 的超时时间。

同样基于安全考虑做的变更还有监听的主机名。现在默认只监听本地 lo 网卡上。所以正式环境上需要修改配置为监听具体的网卡。

```
network.host: "192.168.0.2"
discovery.zen.minimum_master_nodes: 3
discovery.zen.ping_timeout: 100s
discovery.zen.fd.ping_timeout: 100s
discovery.zen.ping.unicast.hosts: ["10.19.0.97","10.19.0.98","10.19.0.99","10.19.0.100"]
```

上面的配置中，两个 timeout 可能会让人有所迷惑。这里的 **fd** 是 fault detection 的缩写。也就是说：

- discovery.zen.ping_timeout 参数仅在加入或者选举 master 主节点的时候才起作用；
- discovery.zen.fd.ping_timeout 参数则在稳定运行的集群中，master 检测所有节点，以及节点检测 master 是否畅通时长期有用。

既然是长期有用，自然还有运行间隔和重试的配置，也可以根据实际情况调整：

```
discovery.zen.fd.ping_interval: 10s
discovery.zen.fd.ping_retries: 10
```

增删改查

增删改查是数据库的基础操作方法。ES 虽然不是数据库，但是很多场合下，都被人们当做一个文档型 NoSQL 数据库在使用，原因自然是因为在接口和分布式架构层面的相似性。虽然在 Elastic Stack 场景下，数据的写入和查询，分别由 Logstash 和 Kibana 代劳，作为测试、调研和排错时的基本功，还是需要了解一下 ES 的增删改查用法的。

数据写入

ES 的一大特点，就是全 RESTful 接口处理 JSON 请求。所以，数据写入非常简单：

```
# curl -XPOST http://127.0.0.1:9200/logstash-2015.06.21/testlog
-d '{
    "date" : "1434966686000",
    "user" : "chenlin7",
    "mesg" : "first message into Elasticsearch"
}'
```

命令返回响应结果为：

```
{"_index":"logstash-2015.06.21","_type":"testlog","_id":"AU4ew3h
2nBE6n0qcyVJK","_version":1,"created":true}
```

数据获取

可以看到，在数据写入的时候，会返回该数据的 `_id` 。这就是后续用来获取数据的关键：

```
# curl -XGET http://127.0.0.1:9200/logstash-2015.06.21/testlog/A
U4ew3h2nBE6n0qcyVJK
```

命令返回响应结果为：

```
{"_index":"logstash-2015.06.21","_type":"testlog","_id":"AU4ew3h
2nBE6n0qcyVJK","_version":1,"found":true,"_source":{
    "date" : "1434966686000",
    "user" : "chenlin7",
    "mesg" : "first message into Elasticsearch"
}}
```

这个 `_source` 里的内容，正是之前写入的数据。

如果觉得这个返回看起来有点太过麻烦，可以使用 `curl -XGET http://127.0.0.1:9200/logstash-2015.06.21/testlog/AU4ew3h2nBE6n0qcyVJK/_source` 来指明只获取源数据部分。

更进一步的，如果你只想看数据中的一部分字段内容，可以使用 `curl -XGET http://127.0.0.1:9200/logstash-2015.06.21/testlog/AU4ew3h2nBE6n0qcyVJK?fields=user,mesg` 来指明获取字段，结果如下：

```
{"_index":"logstash-2015.06.21","_type":"testlog","_id":"AU4ew3h
2nBE6n0qcyVJK","_version":1,"found":true,"fields":{"user":["chen
lin7"],"mesg":["first message into Elasticsearch"]}}
```

数据删除

要删除数据，修改发送的 HTTP 请求方法为 DELETE 即可：

```
# curl -XDELETE http://127.0.0.1:9200/logstash-2015.06.21/testlo
g/AU4ew3h2nBE6n0qcyVJK
```

删除不单针对单条数据，还可以删除整个整个索引。甚至可以用通配符。

```
# curl -XDELETE http://127.0.0.1:9200/logstash-2015.06.0*
```

在 Elasticsearch 2.x 之前，可以通过查询语句删除，也可以删除某个 `_type` 内的数据。现在都已经不再内置支持，改为 `Delete by Query` 插件。因为这种方式本身对性能影响较大！

数据更新

已经写过的数据，同样还是可以修改的。有两种办法，一种是全量提交，即指明 `_id` 再发送一次写入请求。

```
# curl -XPOST http://127.0.0.1:9200/logstash-2015.06.21/testlog/
AU4ew3h2nBE6n0qcyVJK -d '{
    "date" : "1434966686000",
    "user" : "chenlin7",
    "mesg" " "first message into Elasticsearch but version 2"
}'
```

另一种是局部更新，使用 `/_update` 接口：

```
# curl -XPOST 'http://127.0.0.1:9200/logstash-2015.06.21/testlog
/AU4ew3h2nBE6n0qcyVJK/_update' -d '{
    "doc" : {
        "user" : "someone"
    }
}'
```

或者

```
# curl -XPOST 'http://127.0.0.1:9200/logstash-2015.06.21/testlog
/AU4ew3h2nBE6n0qcyVJK/_update' -d '{
    "script" : "ctx._source.user = \"someone\""
}'
```

搜索请求

上节介绍的，都是针对单条数据的操作。在 ES 环境中，更多的是搜索和聚合请求。在 5.0 之前版本中，数据获取和数据搜索甚至有极大的区别：刚写入的数据，可以通过 translog 立刻获取；但是却要等到 refresh 成为一个 segment 后，才能被搜索到。从 5.0 版本开始，Elasticsearch 稍作了改动，不再维护 doc-id 到 translog offset 的映射关系，一旦 GET 请求到这个还不能搜到的数据，就强制 refresh 出来 segment，这样就可以搜索了。这个改动降低了数据获取的性能，但是节省了不少内存，减少了 young GC 次数，对写入性能的提升是很有好处的。

本节介绍 ES 的搜索语法。

全文搜索

ES 对搜索请求，有简易语法和完整语法两种方式。简易语法作为以后在 Kibana 上最常用的方式，一定是需要学会的。而在命令行里，我们可以通过最简单的方式来做到。还是上节输入的数据：

```
# curl -XGET http://127.0.0.1:9200/logstash-2015.06.21/testlog/_
search?q=first
```

可以看到返回结果：

```
{"took":240,"timed_out":false,"_shards":{"total":27,"successful"
:27,"failed":0},"hits":{"total":1,"max_score":0.11506981,"hits":
[{"_index":"logstash-2015.06.21","_type":"testlog","_id":"AU4ew3
h2nBE6n0qcyVJK","_score":0.11506981,"_source":{
    "date" : "1434966686000",
    "user" : "chenlin7",
    "mesg" : "first message into Elasticsearch"
}}]}}
```

还可以用下面语句搜索，结果是一样的。

```
# curl -XGET http://127.0.0.1:9200/logstash-2015.06.21/testlog/_
search?q=user:"chenlin7"
```

querystring 语法

上例中， `?q=` 后面写的，就是 querystring 语法。鉴于这部分内容会在 Kibana 上经常使用，这里详细解析一下语法：

- 全文检索：直接写搜索的单词，如上例中的 `first` ；
- 单字段的全文检索：在搜索单词之前加上字段名和冒号，比如如果知道单词 first 肯定出现在 mesg 字段，可以写作 `mesg:first` ；
- 单字段的精确检索：在搜索单词前后加双引号，比如 `user:"chenlin7"` ；
- 多个检索条件的组合：可以使用 `NOT` , `AND` 和 `OR` 来组合检索，注意必须是大写。比如 `user:("chenlin7" OR "chenlin") AND NOT mesg:first` ；
- 字段是否存在： `_exists_:user` 表示要求 user 字段存在， `_missing_:user` 表示要求 user 字段不存在；
- 通配符：用 `?` 表示单字母， `*` 表示任意个字母。比如 `fir?t mess*` ；
- 正则：需要比通配符更复杂一点的表达式，可以使用正则。比如 `mesg:/mes{2}ages?/` 。注意 ES 中正则性能很差，而且支持的功能也不是特别强大，尽量不要使用。ES 支持的正则语法见：https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-regexp-query.html#regexp-syntax ；
- 近似搜索：用 `~` 表示搜索单词可能有一两个字母写的不对，请 ES 按照相似度返回结果。比如 `frist~` ；
- 范围搜索：对数值和时间，ES 都可以使用范围搜索，比如： `rtt:>300` ， `date:["now-6h" TO "now"}` 等。其中， `[]` 表示端点数值包含在范围内， `{}` 表示端点数值不包含在范围内；

完整语法

ES 支持各种类型的检索请求，除了可以用 querystring 语法表达的以外，还有很多其他类型，具体列表和示例可参见：https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-queries.html 。

作为最简单和常用的示例，这里展示一下 term query 的写法，相当于 querystring 语法中的 `user:"chenlin7"` ：

```
# curl -XGET http://127.0.0.1:9200/_search -d '
{
  "query": {
    "term": {
      "user": "chenlin7"
    }
  }
}'
```

聚合请求

在检索范围确定之后，ES 还支持对结果集做聚合查询，返回更直接的聚合统计结果。在 ES 1.0 版本之前，这个接口叫 Facet，1.0 版本之后，这个接口改为 Aggregation。

Kibana 分别在 v3 中使用 Facet，v4 中使用 Aggregation。不过总的来说，Aggregation 是 Facet 接口的强化升级版本，我们直接了解 Aggregation 即可。本书后续章节也会介绍如何在 Kibana 的 v3 版本中使用 aggregation 接口做二次开发。

堆叠聚合示例

在 Elasticsearch 1.x 系列中，aggregation 分为 bucket 和 metric 两种，分别用作词元划分和数值计算。而其中的 bucket aggregation，还支持在自身结果集的基础上，叠加新的 aggregation。这就是 aggregation 比 facet 最领先的地方。比如实现一个时序百分比统计，在 facet 接口就无法直接完成，而在 aggregation 接口就很简单了：

```
# curl -XPOST 'http://127.0.0.1:9200/logstash-2015.06.22/_search
?size=0&pretty' -d'{
    "aggs" : {
        "percentile_over_time" : {
            "date_histogram" : {
                "field"    : "@timestamp",
                "interval" : "1h"
            },
            "aggs" : {
                "percentile_one_time" : {
                    "percentiles" : {
                        "field" : "requesttime"
                    }
                }
            }
        }
    }
}'
```

得到结果如下：

```
{
  "took" : 151595,
  "timed_out" : false,
  "_shards" : {
    "total" : 81,
    "successful" : 81,
    "failed" : 0
  },
  "hits" : {
    "total" : 3307142043,
    "max_score" : 1.0,
    "hits" : [ ]
  },
  "aggregations" : {
    "percentile_over_time" : {
      "buckets" : [ {
        "key_as_string" : "22/Jun/2015:22:00:00 +0000",
        "key" : 1435010400000,
```

```
        "doc_count" : 459273981,
        "percentile_one_time" : {
          "values" : {
            "1.0" : 0.004,
            "5.0" : 0.006,
            "25.0" : 0.023,
            "50.0" : 0.035,
            "75.0" : 0.08774675719725569,
            "95.0" : 0.25732934416125663,
            "99.0" : 0.7508899754871812
          }
        }
      }, {
        "key_as_string" : "23/Jun/2015:00:00:00 +0000",
        "key" : 1435017600000,
        "doc_count" : 768620219,
        "percentile_one_time" : {
          "values" : {
            "1.0" : 0.004,
            "5.0" : 0.007000000000000001,
            "25.0" : 0.025,
            "50.0" : 0.03987809503972864,
            "75.0" : 0.10297843567746187,
            "95.0" : 0.30047269327062875,
            "99.0" : 1.015495933753329
          }
        }
      }, {
        "key_as_string" : "23/Jun/2015:02:00:00 +0000",
        "key" : 1435024800000,
        "doc_count" : 849467060,
        "percentile_one_time" : {
          "values" : {
            "1.0" : 0.004,
            "5.0" : 0.008,
            "25.0" : 0.027000000000000003,
            "50.0" : 0.0439999899006102,
            "75.0" : 0.1160416197625958,
            "95.0" : 0.3383140614483838,
            "99.0" : 1.0275839684542212
```

```
            }
          }
        } ]
      }
    }
  }
```

管道聚合示例

在 Elasticsearch 2.x 中，新增了 pipeline aggregation 类型。可以在已有 aggregation 返回的数组数据之后，再对这组数值做一次运算。最常见的，就是对时序数据求移动平均值。比如对响应时间做周期为 7，移动窗口为 30，alpha, beta, gamma 参数均为 0.5 的 holt-winters 季节性预测 2 个未来值的请求如下：

```
{
    "aggs" : {
        "my_date_histo" : {
            "date_histogram" : {
                 "field" : "@timestamp",
                 "interval" : "1h"
            },
            "aggs" : {
                "avgtime" : {
                    "avg" : { "field" : "requesttime" }
                },
                "the_movavg" : {
                    "moving_avg" : {
                        "buckets_path" : "avgtime",
                        "window" : 30,
                        "model" : "holt_winters",
                        "predict" : 2,
                        "settings" : {
                            "type" : "mult",
                            "alpha" : 0.5,
                            "beta" : 0.5,
                            "gamma" : 0.5,
                            "period" : 7,
                            "pad" : true
                        }
                    }
                }
            }
        }
    }
}
```

响应如下：

```
{
  "took" : 12,
  "timed_out" : false,
  "_shards" : {
    "total" : 10,
```

```
    "successful" : 10,
    "failed" : 0
  },
  "hits" : {
    "total" : 111331,
    "max_score" : 0.0,
    "hits" : [  ]
  },
  "aggregations" : {
    "my_date_histo" : {
      "buckets" : [ {
        "key_as_string" : "2015-12-24T02:00:00.000Z",
        "key" : 1450922400000,
        "doc_count" : 1462,
        "avgtime" : {
          "value" : 508.25649794801643
        }
      }, {
        ...
      }, {
        "key_as_string" : "2015-12-24T17:00:00.000Z",
        "key" : 1450976400000,
        "doc_count" : 1664,
        "avgtime" : {
          "value" : 504.7067307692308
        },
        "the_movavg" : {
          "value" : 500.9766851760192
        }
      }, {
        ...
      }, {
        "key_as_string" : "2015-12-25T09:00:00.000Z",
        "key" : 1451034000000,
        "doc_count" : 0,
        "the_movavg" : {
          "value" : 493.9519632950849,
          "value_as_string" : "1970-01-01T00:00:00.493Z"
        }
    } ]
```

```
    }
  }
```

可以看到，在第一个移动窗口还没满足之前，是没有移动平均值的；而在实际数据已经结束以后，虽然没有平均值了，但是预测的移动平均值却还有数。

buckets_path 语法

由于 aggregation 是有堆叠层级关系的，所以 pipeline aggregation 在引用 metric aggregation 时也就会涉及到层级的问题。在上例中， `the_movavg` 和 `avgtime` 是同一层级，所以 `buckets_path` 直接写 `avgtime` 即可。但是如果我们把 `the_movavg` 上提一层，跟 `my_date_histo` 同级，这个 `buckets_path` 怎么写才行呢？

```
"buckets_path" : "my_date_histo>avgtime"
```

如果用的是返回的数值有多个值的聚合，比如 `percentiles` 或者 `extended_stats` ，则是：

```
"buckets_path" : "percentile_over_time>percentile_one_time.95"
```

ES 目前能支持的聚合请求列表，参见：https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations.html。

See Also

Holt Winters 预测算法，见：https://en.wikipedia.org/wiki/Holt-Winters。其在运维领域最著名的运用是 RRDtool 中的 HWPREDICT。

search 请求参数

- from

从索引的第几条数据开始返回，默认是 0；

- size

返回多少条数据，默认是 10。

注意：Elasticsearch 集群实际是需要给 coordinate node 返回 `shards number * (from + size)` 条数据，然后在单机上进行排序，最后给客户端返回这个 size 大小的数据的。所以请谨慎使用 from 和 size 参数。

此外，Elasticsearch 2.x 还新增了一个索引级别的动态控制配置项：`index.max_result_window`，默认为 10000。即 `from + size` 大于 10000 的话，Elasticsearch 直接拒绝掉这次请求不进行具体搜索，以保护节点。

另外，Elasticsearch 2.x 还提供了一个小优化：当设置 `"size":0` 时，自动改变 `search_type` 为 count。跳过搜索过程的 fetch 阶段。

- timeout

coordinate node 等待超时时间。到达该阈值后，coordinate node 直接把当前收到的数据返回给客户端，不再继续等待 data node 后续的返回了。

注意：这个参数只是为了配合客户端程序，并不能取消掉 **data node** 上搜索任务还在继续运行和占用资源。

- terminate_after

各 data node 上，扫描单个分片时，找到多少条记录后，就认为足够了。这个参数可以切实保护 data node 上搜索任务不会长期运行和占用资源。但是也就意味着搜索范围没有覆盖全部索引，是一个抽样数据。准确率是不好判断的。

- request_cache

各 data node 上，在分片级别，对请求的响应(仅限于 `hits.total` 数值、aggregation 和 suggestion 的结果集)做的缓存。注意：这个缓存的键值要求很严格，请求的 JSON 必须一字不易，缓存才能命中。

另外，`request_cache` 参数不能写在请求 JSON 里，只能以 URL 参数的形式存在。示例如下：

```
curl -XPOST http://localhost:9200/_search?request_cache=true -d
'
{
    "size" : 0,
    "timeout" : "120s",
    "terminate_after" : 1000000,
    "query" : { "match_all" : {} },
    "aggs" : { "terms" : { "terms" : { "field" : "keyname" } } }
}
'
```

script

Elasticsearch 中，可以使用自定义脚本扩展功能。包括评分、过滤函数和聚合字段等方面。内置脚本引擎历经 MVEL、Groovy、Lucene expression 的变换后，Elastic.co 最终决定实现一个自己专用的 Painless 脚本语言，并在 5.0 版正式发布。

作为 Elastic Stack 场景，我们只介绍在聚合字段方面使用 script 的方式。

动态提交

最简单易用的方式，就是在正常的请求体中，把 `field` 换成 `script` 提交。比如一个标准的 terms agg 改成 script 方式，写法如下：

```
# curl 127.0.0.1:9200/logstash-2015.06.29/_search -d '{
    "aggs" : {
        "clientip_top10" : {
            "terms" : {
                "script" : {
                    "lang" : "painless",
                    "inline" : "doc['clientip'].value"
                }
            }
        }
    }
}'
```

在 script 中，有三种方式引用数据：`doc['clientip'].value`、`_field['clientip'].value` 和 `_source.clientip`。其区别在于：

- `doc[].value` 读取 doc value 内的数据；
- `_field[]` 读取 field 设置 `"store":true` 的存储内容；
- `_source.obj.attr` 读取 `_source` 的 JSON 内容。

这也意味着，前者必须读取的是最终的词元字段数据，而后者可以返回任意的数据结构。

注意：如果有分词，且未禁用 fielddata 的话， `doc[].value` 读取到的是分词后的数据。所以请注意使用 `doc['clientip.keyword'].value` 写法。

固定文件

为了和动态提交的语法有区别，调用固定文件的写法如下：

```
# curl 127.0.0.1:9200/logstash-2015.06.29/_search -d '{
    "aggs" : {
        "clientip_subnet_top10" : {
            "terms" : {
                "script" : {
                   "file" : "getvalue",
                    "lang" : "groovy",
                    "params" : {
                        "fieldname": "clientip.keyword",
                        "pattern": "^((?:\d{1,3}\.?){3})\.\d{1,3
}$"
                    }
                }
            }
        }
    }
}'
```

上例要求在 ES 集群的所有数据节点上，都保存有一个 `/etc/elasticsearch/scripts/getvalue.groovy` 文件，并且该脚本文件可以接收 `fieldname` 和 `pattern` 两个变量。试举例如下：

```
#!/usr/bin/env groovy
matcher = ( doc[fieldname].value =~ /${pattern}/ )
if (matcher.matches()) {
    matcher[0][1]
}
```

注意：ES 进程默认每分钟扫描一次 `/etc/elasticsearch/scripts/` 目录，并尝试加载该目录下所有文件作为 script。所以，不要在该目录内做文件编辑等工作，不要分发 .svn 等目录到生成环境，这些临时或者隐藏文件都会被 ES 进程加载然后报错。

其他语言

ES 支持通过插件方式，扩展脚本语言的支持，目前默认自带的语言包括：

- painless
- lucene expression
- groovy
- mustache

而 github 上目前已有以下语言插件支持，基本覆盖了所有 JVM 上的可用语言：

- https://github.com/elastic/elasticsearch-lang-mvel
- https://github.com/elastic/elasticsearch-lang-javascript
- https://github.com/elastic/elasticsearch-lang-python
- https://github.com/hiredman/elasticsearch-lang-clojure
- https://github.com/felipehummel/elasticsearch-lang-scala
- https://github.com/fcheung/elasticsearch-jruby

reindex

Elasticsearch 本身不提供对索引的 rename，mapping 的 alter 等操作。所以，如果有需要对全索引数据进行导出，或者修改某个已有字段的 mapping 设置等情况下，我们只能通过 scroll API 导出全部数据，然后重新做一次索引写入。这个过程，叫做 reindex。

之前完成这个过程只能自己写程序或者用 logstash。5.0 中，Elasticsearch 将这个过程内置为 reindex API，但是要注意：这个接口并没有什么黑科技，其本质仅仅是将这段相同逻辑的代码预置分发而已。如果有复杂的数据变更操作等细节需求，依然需要自己编程完成。

下面分别给出这三种方法的示例：

Perl 客户端

Elastic 官方提供各种语言的客户端库，其中，Perl 库提供了对 reindex 比较方便的写法和示例。通过 `cpanm Search::Elasticsearch` 命令安装库完毕后，使用以下程序即可：

```
use Search::Elasticsearch;

my $es   = Search::Elasticsearch->new(
    nodes => ['192.168.0.2:9200']
);
my $bulk = $es->bulk_helper(
    index   => 'new_index',
);

$bulk->reindex(
    source  => {
        index       => 'old_index',
        size        => 500,         # default
        search_type => 'scan'       # default
    }
);
```

Logstash 做 reindex

在最新版的 Logstash 中，对 logstash-input-elasticsearch 插件做了一定的修改，使得通过 logstash 完成 reindex 成为可能。

reindex 操作的 logstash 配置如下：

```
input {
  elasticsearch {
    hosts => [ "192.168.0.2" ]
    index => "old_index"
    size => 500
    scroll => "5m"
    docinfo => true
  }
}
output {
  elasticsearch {
    hosts => [ "192.168.0.3" ]
    index => "%{[@metadata][_index]}"
    document_type => "%{[@metadata][_type]}"
    document_id => "%{[@metadata][_id]}"
  }
}
```

如果你做 reindex 的源索引并不是 logstash 记录的内容，也就是没有 `@timestamp`，`@version` 这两个 logstash 字段，那么可以在上面配置中添加一段 filter 配置，确保前后索引字段完全一致：

```
filter {
  mutate {
    remove_field => [ "@timestamp", "@version" ]
  }
}
```

reindex API

简单的 reindex，可以很容易的完成：

```
curl -XPOST http://localhost:9200/_reindex -d '
{
  "source": {
    "index": "logstash-2016.10.29"
  },
  "dest": {
    "index": "logstash-new-2016.10.29"
  }
}'
```

复杂需求，也能通过配合其他 API，比如 script、pipeline 等来满足一些，下面举一个复杂的示例：

```
curl -XPOST http://localhost:9200/_reindex?requests_per_second=1
0000 -d '
{
  "source": {
    "remote": {
      "host": "http://192.168.0.2:9200",
    },
    "index": "metricbeat-*",
    "query": {
      "match": {
        "host": "webserver"
      }
    }
  },
  "dest": {
    "index": "metricbeat",
    "pipeline": "ingest-rule-1"
  },
  "script": {
    "lang": "painless",
    "inline": "ctx._index = 'metricbeat-' + (ctx._index.substrin
g('metricbeat-'.length(), ctx._index.length())) + '-1'"
  }
}'
```

上面这个请求的作用，是将来自 192.168.0.2 集群的 metricbeat-2016.10.29 索引中，有关 `host:webserver` 的数据，读取出来以后，经过 localhost 集群的 `ingest-rule-1` 规则处理，在写入 localhost 集群的 metricbeat-2016.10.29-1 索引中。

注意：读取远端集群数据需要先配置对应的 `reindex.remote.whitelist:192.168.0.2:9200` 到 elasticsearch.yml 的白名单里。

通过 reindex 接口运行的任务可以通过同样是 5.0 新引入的任务管理接口进行取消、修改等操作。详细介绍见后续任务管理章节。

批量提交

在 CRUD 章节，我们已经知道 ES 的数据写入是如何操作的了。喜欢自己动手的读者可能已经迫不及待的自己写了程序开始往 ES 里写数据做测试。这时候大家会发现：程序的运行速度非常一般，即使 ES 服务运行在本机，一秒钟大概也就能写入几百条数据。

这种速度显然不是 ES 的极限。事实上，每条数据经过一次完整的 HTTP POST 请求和 ES indexing 是一种极大的性能浪费，为此，ES 设计了批量提交方式。在数据读取方面，叫 mget 接口，在数据变更方面，叫 bulk 接口。mget 一般常用于搜索时 ES 节点之间批量获取中间结果集，对于 Elastic Stack 用户，更常见到的是 bulk 接口。

bulk 接口采用一种比较简朴的数据积累格式，示例如下：

```
# curl -XPOST http://127.0.0.1:9200/_bulk -d'
{ "create" : { "_index" : "test", "_type" : "type1"  } }
{ "field1" : "value1" }
{ "delete" : { "_index" : "test", "_type" : "type1" } }
{ "index" : { "_index" : "test", "_type" : "type1", "_id" : "1"
} }
{ "field1" : "value2" }
{ "update" : {"_id" : "1", "_type" : "type1", "_index" : "test"}
 }
{ "doc" : {"field2" : "value2"} }
'
```

格式是，每条 JSON 数据的上面，加一行描述性的元 JSON，指明下一行数据的操作类型，归属索引信息等。

采用这种格式，而不是一般的 JSON 数组格式，是因为接收到 bulk 请求的 ES 节点，就可以不需要做完整的 JSON 数组解析处理，直接按行处理简短的元 JSON，就可以确定下一行数据 JSON 转发给哪个数据节点了。这样，一个固定内存大小的 network buffer 空间，就可以反复使用，又节省了大量 JVM 的 GC。

事实上，产品级的 logstash、rsyslog、spark 都是默认采用 bulk 接口进行数据写入的。对于打算自己写程序的读者，建议采用 Perl 的 `Search::Elasticsearch::Bulk` 或者 Python 的 `elasticsearch.helpers.*` 库。

bulk size

在配置 bulk 数据的时候，一般需要注意的就是请求体大小(bulk size)。

这里有一点细节上的矛盾，我们知道，HTTP 请求，是可以通过 HTTP 状态码 *100 Continue* 来持续发送数据的。但对于 ES 节点接收 HTTP 请求体的 *Content-Length* 来说，是按照整个大小来计算的。所以，首先，要确保 bulk 数据不要超过 `http.max_content_length` 设置。

那么，是不是尽量让 bulk size 接近这个数值呢？当然不是。

依然是请求体的问题，因为请求体需要全部加载到内存，而 JVM Heap 一共就那么多(按 31GB 算)，过大的请求体，会挤占其他线程池的空间，反而导致写入性能的下降。

再考虑网卡流量，磁盘转速的问题，所以一般来说，建议 bulk 请求体的大小，在 15MB 左右，通过实际测试继续向上探索最合适的设置。

注意：这里说的 15MB 是请求体的字节数，而不是程序里里设置的 bulk size。bulk size 一般指数据的条目数。不要忘了，bulk 请求体中，每条数据还会额外带上一行元 JSON。

以 logstash 默认的 `bulk_size => 5000` 为例，假设单条数据平均大小 200B ，一次 bulk 请求体的大小就是 1.5MB。那么我们可以尝试 `bulk_size => 50000` ；而如果单条数据平均大小是 20KB，一次 bulk 大小就是 100MB，显然超标了，需要尝试下调至 `bulk_size => 500` 。

gateway

gateway 是 ES 设计用来长期存储索引数据的接口。一般来说，大家都是用本地磁盘来存储索引数据，即 `gateway.type` 为 `local` 。

数据恢复中，有很多策略调整我们已经在之前分片控制小节讲过。除开分片级别的控制以外，gateway 级别也还有一些可优化的地方：

- gateway.recover_after_nodes 该参数控制集群在达到多少个节点的规模后，才开始数据恢复任务。这样可以避免集群自动发现的初期，分片不全的问题。
- gateway.recover_after_time 该参数控制集群在达到上条配置设置的节点规模后，再等待多久才开始数据恢复任务。
- gateway.expected_nodes 该参数设置集群的预期节点总数。在达到这个总数后，即认为集群节点已经完全加载，即可开始数据恢复，不用再等待上条设置的时间。

注意：gateway 中说的节点，仅包括主节点和数据节点，纯粹的 client 节点是不算在内的。如果你有更明确的选择，也可以按需求写：

- gateway.recover_after_data_nodes
- gateway.recover_after_master_nodes
- gateway.expected_data_nodes
- gateway.expected_master_nodes

共享存储上的影子副本

虽然 ES 对 gateway 使用 NFS，iscsi 等共享存储的方式极力反对，但是对于较大量级的索引的副本数据，ES 从 1.5 版本开始，还是提供了一种节约成本又不特别影响性能的方式：影子副本(shadow replica)。

首先，需要在集群各节点的 `elasticsearch.yml` 中开启选项：

```
node.enable_custom_paths: true
```

同时，确保各节点使用相同的路径挂载了共享存储，且目录权限为 Elasticsearch 进程用户可读可写。

然后，创建索引：

```
# curl -XPUT 'http://127.0.0.1:9200/my_index' -d '
{
    "index" : {
        "number_of_shards" : 1,
        "number_of_replicas" : 4,
        "data_path": "/var/data/my_index",
        "shadow_replicas": true
    }
}'
```

针对 shadow replicas ，ES 节点不会做实际的索引操作，而是单纯的每次 flush 时，把 segment 内容 fsync 到共享存储磁盘上。然后 refresh 让其他节点能够搜索该 segment 内容。

如果你已经决定把数据放到共享存储上了，采用 shadow replicas 还是有一些好处的：

1. 可以帮助你节省一部分不必要的多副本分片的数据写入压力；
2. 在节点出现异常，需要在其他节点上恢复副本数据的时候，可以避免不必要的网络数据拷贝。

但是请注意：主分片节点还是要承担一个副本的写入过程，并不像 Lucene 的 FileReplicator 那样通过复制文件完成，所以达不到完全节省 CPU 的效果。

shadow replicas 只是一个在某些特定环境下有用的方式。在资源允许的情况下，还是应该使用 local gateway。而另外采用 snapshot 接口来完成数据长期备份到 HDFS 或其他共享存储的需要。

集群状态维护

我们都知道，ES 中的 master 跟一般 MySQL、Hadoop 的 master 是不一样的。它即不是写入流量的唯一入口，也不是所有数据的元信息的存放地点。所以，一般来说，ES 的 master 节点负载很轻，集群性能是可以近似认为随着 data 节点的扩展线性提升的。

但是，上面这句话并不是完全正确的。

ES 中有一件事情是只有 master 节点能管理的，这就是集群状态(cluster state)。

集群状态中包括以下信息：

- 集群层面的设置
- 集群内有哪些节点
- 各索引的设置，映射，分析器和别名等
- 索引内各分片所在的节点位置

这些信息在集群的任意节点上都存放着，你也可以通过 `/_cluster/state` 接口直接读取到其内容。注意这最后一项信息，之前我们已经讲过 ES 怎么通过简单地取余知道一条数据放在哪个分片里，加上现在集群状态里又记载了分片在哪个节点上，那么，整个集群里，任意节点都可以知道一条数据在哪个节点上存储了。所以，数据读写才可以发送给集群里任意节点。

至于修改，则只能由 master 节点完成！显然，集群状态里大部分内容是极少变动的，唯独有一样除外——索引的映射。因为 ES 的 schema-less 特性，我们可以任意写入 JSON 数据，所以索引中随时可能增加新的字段。这个时候，负责容纳这条数据的主分片所在的节点，会暂停写入操作，将字段的映射结果传递给 master 节点；master 节点合并这段修改到集群状态里，发送新版本的集群状态到集群的所有节点上。然后写入操作才会继续。一般来说，这个操作是在一二十毫秒内就可以完成，影响也不大。

但是也有一些情况会是例外。

批量新索引创建

在较大规模的 Elastic Stack 应用场景中，这是比较常见的一个情况。因为 Elastic Stack 建议采用日期时间作为索引的划分方式，所以定时(一般是每天)，会统一产生一批新的索引。而前面已经讲过，ES 的集群状态每次更新都是阻塞式的发布到全部节点上以后，节点才能继续后续处理。

这就意味着，如果在集群负载较高的时候，批量新建新索引，可能会有一个显著的阻塞时间，无法写入任何数据。要等到全部节点同步完成集群状态以后，数据写入才能恢复。

不巧的是，中国使用的是北京时间，UTC +0800。也就是说，默认的 Elastic Stack 新建索引时间是在早上 8 点。这个时间点一般日志写入量已经上涨到一定水平了(当然，晚上 0 点的量其实也不低)。

对此，可以通过定时任务，每天在最低谷的早上三四点，提前通过 POST mapping 的方式，创建好之后几天的索引。就可以避免这个问题了。

如果你的日志是比较严重的非结构化数据，这个问题在 **2.0** 版本后会变得更加严重。 Elasticsearch 从 2.0 版本开始，对 mapping 更新做了重构。为了防止字段类型冲突和减少 master 定期下发全量 cluster state 导致的大流量压力，新的实现和旧实现的区别在：

- 过去：每次 bulk 请求，本地生成索引后，将更新的 mapping，按照 `_type` 为单位构成 mapping 更新请求发给 master；
- 现在：每次 bulk 请求，遍历每条数据，将每条数据要更新的 mapping，都单独发给 master，等到 master 通知完全集群，本地才能生成这一条数据的索引。

也就是说，一旦你日志中字段数量较多，在新创建索引的一段时间内，可能长达几十分钟一直被反复锁死！

过多字段持续更新

这是另一种常见的滥用。在使用 Elastic Stack 处理访问日志时，为了查询更方便，可能会采用 logstash-filter-kv 插件，将访问日志中的每个 URL 参数，都切分成单独的字段。比如一个 "/index.do?uid=1234567890&action=payload" 的 URL 会被转换成如下 JSON：

```
  "urlpath" : "/index.do",
  "urlargs" : {
    "uid" : "1234567890",
    "action" : "payload",
    ...
  }
```

但是，因为集群状态是存在所有节点的内存里的，一旦 URL 参数过多，ES 节点的内存就被大量用于存储字段映射内容。这是一个极大的浪费。如果碰上 URL 参数的键内容本身一直在变动，直接撑爆 ES 内存都是有可能的！

以上是真实发生的事件，开发人员莫名的选择将一个 *UUID* 结果作为 *key* 放在 *URL* 参数里。直接导致 *ES* 集群 *master* 节点全部 *OOM*。

如果你在 ES 日志中一直看到有新的 `updating mapping [logstash-2015.06.01]` 字样出现的话，请郑重考虑一下自己是不是用的上如此细分的字段列表吧。

好，三秒钟过去，如果你确定一定以及肯定还要这么做，下面是一个变通的解决办法。

nested object

用 nested object 来存放 URL 参数的方法稍微复杂，但还可以接受。单从 JSON 数据层面看，新方式的数据结构如下：

```
  "urlargs": [
    { "key": "uid", "value": "1234567890" },
    { "key": "action", "value": "payload" },
    ...
  ]
```

没错，看起来就是一个数组。但是 JSON 数组在 ES 里是有两种处理方式的。

如果直接写入数组，ES 在实际索引过程中，会把所有内容都平铺开，变成 **Arrays of Inner Objects**。整条数据实际类似这样的结构：

```
{
  "urlpath" : ["/index.do"],
  "urlargs.key" : ["uid", "action", ...],
  "urlargs.value" : ["1234567890", "payload", ...]
```

这种方式最大的问题是，当你采用 `urlargs.key:"uid" AND urlargs.value:"0987654321"` 语句意图搜索一个 uid=0987654321 的请求时，实际是整个 URL 参数中任意一处 value 为 0987654321 的，都会命中。

要想达到正确搜索的目的，需要在写入数据之前，指定 urlargs 字段的映射类型为 nested object。命令如下：

```
curl -XPOST http://127.0.0.1:9200/logstash-2015.06.01/_mapping -
d '{
  "accesslog" : {
    "properties" : {
      "urlargs" : {
        "type" : "nested",
        "properties" : {
            "key" : { "type" : "string", "index" : "not_analyzed
", "doc_values" : true },
            "value" : { "type" : "string", "index" : "not_analyz
ed", "doc_values" : true }
        }
      }
    }
  }
}'
```

这样，数据实际是类似这样的结构：

```
{
  "urlpath" : ["/index.do"],
},
{
  "urlargs.key" : ["uid"],
  "urlargs.value" : ["1234567890"],
},
{
  "urlargs.key" : ["action"],
  "urlargs.value" : ["payload"],
}
```

当然，nested object 节省字段映射的优势对应的是它在使用的复杂。Query 和 Aggs 都必须使用专门的 nested query 和 nested aggs 才能正确读取到它。

nested query 语法如下：

```
curl -XPOST http://127.0.0.1:9200/logstash-2015.06.01/accesslog/
_search -d '
{
  "query": {
    "bool": {
      "must": [
        { "match": { "urlpath" : "/index.do" }},
        {
          "nested": {
            "path": "urlargs",
            "query": {
              "bool": {
                "must": [
                  { "match": { "urlargs.key": "uid" }},
                  { "match": { "urlargs.value": "1234567890" }}
                ]
        }}}}
      ]
}}}'
```

nested aggs 语法如下：

```
curl -XPOST http://127.0.0.1:9200/logstash-2015.06.01/accesslog/
_search -d '
{
  "aggs": {
    "topnuid": {
      "nested": {
        "path": "urlargs"
      },
      "aggs": {
        "uid": {
          "filter": {
            "term": {
              "urlargs.key": "uid",
            }
          },
          "aggs": {
            "topn": {
              "terms": {
                "field": "urlargs.value"
              }
            }
          }
        }
      }
    }
  }
}'
```

缓存

ES 内针对不同阶段，设计有不同的缓存。以此提升数据检索时的响应性能。主要包括节点层面的 filter cache 和分片层面的 request cache。下面分别讲述。

filter cache

ES 的 query DSL 在 2.0 版本之前分为 query 和 filter 两种，很多检索语法，是同时存在 query 和 filter 里的。比如最常用的 term、prefix、range 等。怎么选择是使用 query 还是 filter 成为很多用户头疼的难题。于是从 2.0 版本开始，ES 干脆合并了 filter 统一归为 query。但是具体的检索语法本身，依然有 query 和 filter 上下文的区别。ES 依靠这个上下文判断，来自动决定是否启用 filter cache。

query 跟 filter 上下文的区别，简单来说：

- query 是要相关性评分的，filter 不要；
- query 结果无法缓存，filter 可以。

所以，选择也就出来了：

- 全文搜索、评分排序，使用 query；
- 是非过滤，精确匹配，使用 filter。

不过我们要怎么写，才能让 ES 正确判断呢？看下面这个请求：

```
# curl -XGET http://127.0.0.1:9200/_search -d '
{
    "query": {
        "bool": {
            "must_not": [
                { "match": { "title": "Search" } }
            ],
            "must": [
                { "match": { "content": "Elasticsearch" } }
            ],
            "filter": [
                { "term":  { "status": "published" } },
                { "range": { "publish_date": { "gte": "2015-01-0
1" } } }
            ]
        }
    }
}'
```

在这个请求中，

1. ES 先看到一个 **query**，那么进入 query 上下文。
2. 然后在 **bool** 里看到一个 **must_not**，那么改进入 filter 上下文，这个有关 title 字段的查询不参与评分。
3. 然后接着是一个 **must** 的 **match**，这个又属于 query 上下文，这个有关 content 字段的查询会影响评分。
4. 最后碰到 **filter**，还属于 filter 上下文，这个有关 status 和 publish_date 字段的查询不参与评分。

需要注意的是，filter cache 是节点层面的缓存设置，每个节点上所有数据在响应请求时，是共用一个缓存空间的。当空间用满，按照 LRU 策略淘汰掉最冷的数据。

可以用 `indices.cache.filter.size` 配置来设置这个缓存空间的大小，默认是 JVM 堆的 10%，也可以设置一个绝对值。注意这是一个静态值，必须在 `elasticsearch.yml` 中提前配置。

shard request cache

ES 还有另一个分片层面的缓存，叫 shard request cache。5.0 之前的版本中，request cache 的用途并不大，因为 query cache 要起作用，还有几个先决条件：

1. 分片数据不再变动，也就是对当天的索引是无效的(如果 `refresh_interval` 很大，那么在这个间隔内倒也算有效)；
2. 使用了 `"now"` 语法的请求无法被缓存，因为这个是要即时计算的；
3. 缓存的键是请求的整个 JSON 字符串，整个字符串发生任何字节变动，缓存都无效。

以 Elastic Stack 场景来说，Kibana 里几乎所有的请求，都是有 `@timestamp` 作为过滤条件的，而且大多数是以最近 *N* 小时/分钟这样的选项，也就是说，页面每次刷新，发出的请求 JSON 里的时间过滤部分都是在变动的。query cache 在处理 Kibana 发出的请求时，完全无用。

而 5.0 版本的一大特性，叫 instant aggregation。解决了这个先决条件的一大阻碍。

在之前的版本，Elasticsearch 接收到请求之后，直接把请求原样转发给各分片，由各分片所在的节点自行完成请求的解析，进行实际的搜索操作。所以缓存的键是原始 JSON 串。

而 5.0 的重构后，接收到请求的节点先把请求的解析做完，发送到各节点的是统一拆分修改好的请求，这样就不再担心 JSON 串多个空格啥的了。

其次，上面说的『拆分修改』是怎么回事呢？

比如，我们在 Kibana 里搜索一个最近 7 天(`@timestamp:["now-7d" TO "now"]`)的数据，ES 就可以根据按天索引的判断，知道从 6 天前到昨天这 5 个索引是肯定全覆盖的。那么这个横跨 7 天的 `date range` query 就变成了 5 个 `match_all` query 加 2 个短时间的 `date_range` query。

现在你的仪表盘过 5 分钟自动刷新一次，再提交上来一次最近 7 天的请求，中间这 5 个 `match_all` 就完全一样了，直接从 request cache 返回即可，需要重新请求的，只有两头真正在变动的 `date_range` 了。

注*1*： *`match_all`* 不用遍历倒排索引，比直接查询 *`@timestamp:* `要快很多。* *注2：判断覆盖修改为` match_all `并不是真的按照索引名称，而是 ES 从 2.x 开始提供的` field_stats `接口可以直接获取到` @timestamp` 在本索引内的 max/min 值。当然从概念上如此理解也是可以接受的。*

field_stats 接口

```
curl -XGET "http://localhost:9200/logstash-2016.11.25/_field_stats?fields=timestamp"
```

响应结果如下：

```
{
    "_shards": {
        "total": 1,
        "successful": 1,
        "failed": 0
    },
    "indices": {
        "logstash-2016.11.25": {
            "fields": {
                "timestamp": {
                    "max_doc": 1326564,
                    "doc_count": 564633,
                    "density": 42,
                    "sum_doc_freq": 2258532,
                    "sum_total_term_freq": -1,
                    "min_value": "2008-08-01T16:37:51.513Z",
                    "max_value": "2013-06-02T03:23:11.593Z",
                    "is_searchable": "true",
                    "is_aggregatable": "true"
                }
            }
        }
    }
}
```

和 filter cache 一样，request cache 的大小也是以节点级别控制的，配置项名为 `indices.requests.cache.size` ，其默认值为 `1%` 。

字段数据

字段数据(fielddata)，在 Lucene 中又叫 uninverted index。我们都知道，搜索引擎会使用倒排索引(inverted index)来映射单词到文档的 ID 号。而同时，为了提供对文档内容的聚合，Lucene 还可以在运行时将每个字段的单词以字典序排成另一个 uninverted index，可以大大加速计算性能。

作为一个加速性能的方式，fielddata 当然是被全部加载在内存的时候最为有效。这也是 ES 默认的运行设置。但是，内存是有限的，所以 ES 同时也需要提供对 fielddata 内存的限额方式：

- indices.fielddata.cache.size 节点用于 fielddata 的最大内存，如果 fielddata 达到该阈值，就会把旧数据交换出去。该参数可以设置百分比或者绝对值。默认设置是不限制，所以强烈建议设置该值，比如 `10%` 。
- indices.fielddata.cache.expire 进入 fielddata 内存中的数据多久自动过期。注意，因为 ES 的 fielddata 本身是一种数据结构，而不是简单的缓存，所以过期删除 fielddata 是一个非常消耗资源的操作。ES 官方在文档中特意说明，这个参数绝对绝对不要设置！

Circuit Breaker

Elasticsearch 在 total，fielddata，request 三个层面上都设计有 circuit breaker 以保护进程不至于发生 OOM 事件。在 fielddata 层面，其设置为：

- indices.breaker.fielddata.limit 默认是 JVM 堆内存大小的 60%。注意，为了让设置正常发挥作用，如果之前设置过 `indices.fielddata.cache.size` 的，一定要确保 `indices.breaker.fielddata.limit` 的值大于 `indices.fielddata.cache.size` 的值。否则的话，fielddata 大小一到 limit 阈值就报错，就永远道不了 size 阈值，无法触发对旧数据的交换任务了。

doc values

但是相比较集群庞大的数据量，内存本身是远远不够的。为了解决这个问题，ES 引入了另一个特性，可以对精确索引的字段，指定 fielddata 的存储方式。这个配置项叫： `doc_values` 。

所谓 `doc_values` ，其实就是在 ES 将数据写入索引的时候，提前生成好 fielddata 内容，并记录到磁盘上。因为 fielddata 数据是顺序读写的，所以即使在磁盘上，通过文件系统层的缓存，也可以获得相当不错的性能。

注意：因为 `doc_values` 是在数据写入时即生成内容，所以，它只能应用在精准索引的字段上，因为索引进程没法知道后续会有什么分词器生成的结果。

由于在 Elastic Stack 场景中， `doc_values` 的使用极其频繁，到 Elasticsearch 5.0 以后，这两者的区别被彻底强化成两个不同字段类型： `text` 和 `keyword` 。

```
"myfieldname": {
    "type": "text"
}
```

等同于过去的：

```
    "myfieldname": {
        "type": "string",
        "fielddata": false
    }
```

而

```
"myfieldname": {
    "type": "keyword"
}
```

等同于过去的：

```
    "myfieldname": {
        "type": "string",
        "index": "not_analyzed",
        "doc_values": true
    }
```

也就是说，以后的用户，已经不太需要在意 fielddata 的问题了。不过依然有少数情况，你会需要对分词字段做聚合统计的话，你可以在自己接受范围内，开启这个特性：

```
{
    "mappings": {
        "my_type": {
            "properties": {
                "message": {
                    "type": "text",
                    "fielddata": true,
                    "fielddata_frequency_filter": {
                        "min": 0.1,
                        "max": 1.0,
                        "min_segment_size": 500
                    }
                }
            }
        }
    }
}
```

你可以看到在上面加了一段 `fielddata_frequency_filter` 配置，这个配置是 segment 级别的。上面示例的意思是：只有这个 segment 里的文档数量超过 500 个，而且含有该字段的文档数量占该 segment 里的文档数量比例超过 10% 时，才加载这个 segment 的 fielddata。

下面是一个可能有用的对分词字段做聚合的示例：

```
curl -XPOST 'http://localhost:9200/logstash-2016.07.18/logs/_sea
rch?pretty&terminate_after=10000&size=0' -d '
{
    "aggs": {
        "group": {
            "terms": {
                "field": "punct"
            },
            "aggs": {
                "keyword": {
                    "significant_terms": {
                        "size": 2,
                        "field": "message"
                    },
                    "aggs": {
                        "hit": {
                            "top_hits": {
                                "_source": {
                                    "include": [ "message" ]
                                },
                                "size":1
                            }
                        }
                    }
                }
            }
        }
    }
}'
```

这个示例可以对经过了 `logstash-filter-punct` 插件处理的数据，获取每种 punct 类型日志的关键词和对应的代表性日志原文。其效果类似 Splunk 的事件模式功能：

splunk> 应用: Search & Reporting Administrator 消息 设置 活动 帮助 查找

搜索 数据透视表 报表 告警 仪表板 Search & Reporting

新搜索 另存为 关闭

index="_internal" | 所有时间

✓ 13,365 个事件 (16/05/31 18:22:05.000 之前) No Event Sampling 任务 详细模式

事件 (13,365) 模式 统计信息 可视化

19 模式 基于 13,365 个事件 示例 更小 更大

27.13% <时间戳> INFO Metrics - group=pipeline, name=typing, processor=sendout, cpu_seconds=0.000000, executes=112, cumulative_hits=19594

16.83% <时间戳> INFO Metrics - group=queue, name=typingqueue, max_size_kb=500, current_size_kb=0, current_size=0, largest_size=4, smallest_size=0

15.2% <时间戳> INFO Metrics - group=per_sourcetype_thruput, series="splunkd_ui_access", kbps=0.331398, eps=0.741929, kb=10.273438, ev=23, avg_age=0.739130, max_age=1

7.9% 127.0.0.1 - admin [<时间戳>] "GET /zh-CN/splunkd/__raw/services/messages?output_mode=json&sort_key=timeCreated_epochSecs&sort_dir=desc&count=1000&_=1464689338995 HTTP/1.1" 200 296 "http://localhost:8000/zh-CN/app/search/search?q=search%20index%3D%22_internal%22%20%7C%20typelearner&display.page.search.mode=verbose&dispatch.sample_ratio=1&earliest=&latest=&display.pag

5.18% <时间戳> INFO Metrics - group=deploy-server, name=clients, phonehome, nReceived=0

3.88% <时间戳> INFO Metrics - group=thruput, name=thruput, instantaneous_kbps=1.143293, instantaneous_eps=3.709647, average_kbps=1.073732, total_k_processed=5758.000000, kb=35.442383, ev=115.000000, load_average=2.088379

2.59% <时间戳> INFO Metrics - group=tpool, name=indexertpool, qsize=0, workers=2, qwork_units=0

2.59% <时间戳> INFO Metrics - group=tailingprocessor, name=tailreader0, current_queue_size=0, max_queue_size=2, files_queued=18, new_files_queued=0, fd_cache_size=2

1.56% <时间戳> INFO Metrics - group=search_health_metrics, name=compute_search_quota, compute_search_quota_max_ms=3, compute_search_quota_mean_ms=2.000000

1.38% <时间戳> INFO Metrics - group=search_concurrency, user=admin, active_hist_searches=1, active_realtime_searches=0

1.29% <时间戳> INFO Metrics - group=mpool, max_used_interval=26649, max_used=359383, avg_rsv=440, capacity=536870912, used=2738, rep

预计的事件

3.63K

查看事件

搜索

index="_internal" pipeline

另存为事件类型 创建告警

包括关键字

pipeline

curator

如果经过之前章节的一系列优化之后，数据确实超过了集群能承载的能力，除了拆分集群以外，最后就只剩下一个办法了：清除废旧索引。

为了更加方便的做清除数据，合并 segment，备份恢复等管理任务，Elasticsearch 在提供相关 API 的同时，另外准备了一个命令行工具，叫 curator 。curator 是 Python 程序，可以直接通过 pypi 库安装：

```
pip install elasticsearch-curator
```

注意，是 *elasticsearch-curator* 不是 *curator*。*PyPi* 原先就有另一个项目叫这个名字

参数介绍

和 Elastic Stack 里其他组件一样，curator 也是被 Elastic.co 收购的原开源社区周边。收编之后同样进行了一次重构，命令行参数从单字母风格改成了长单词风格。新版本的 curator 命令可用参数如下：

> Usage: curator [OPTIONS] COMMAND [ARGS]...

Options 包括:

--host TEXT Elasticsearch host. --url_prefix TEXT Elasticsearch http url prefix. --port INTEGER Elasticsearch port. --use_ssl Connect to Elasticsearch through SSL. --http_auth TEXT Use Basic Authentication ex: user:pass --timeout INTEGER Connection timeout in seconds. --master-only Only operate on elected master node. --dry-run Do not perform any changes. --debug Debug mode --loglevel TEXT Log level --logfile TEXT log file --logformat TEXT Log output format [default|logstash]. --version Show the version and exit. --help Show this message and exit.

Commands 包括: alias Index Aliasing allocation Index Allocation bloom Disable bloom filter cache close Close indices delete Delete indices or snapshots open Open indices optimize Optimize Indices replicas Replica Count Per-shard show

Show indices or snapshots snapshot Take snapshots of indices (Backup)

针对具体的 Command，还可以继续使用 `--help` 查看该子命令的帮助。比如查看 *close* 子命令的帮助，输入 `curator close --help` ，结果如下：

```
Usage: curator close [OPTIONS] COMMAND [ARGS]...

  Close indices

Options:
  --help  Show this message and exit.

Commands:
  indices  Index selection.
```

常用示例

在使用 1.4.0 以上版本的 Elasticsearch 前提下，curator 曾经主要的一个子命令 `bloom` 已经不再需要使用。所以，目前最常用的三个子命令，分别是 `close` , `delete` 和 `optimize` ，示例如下：

```
curator --timeout 36000 --host 10.0.0.100 delete indices --older
-than 5 --time-unit days --timestring '%Y.%m.%d' --prefix logsta
sh-mweibo-nginx-
curator --timeout 36000 --host 10.0.0.100 delete indices --older
-than 10 --time-unit days --timestring '%Y.%m.%d' --prefix logst
ash-mweibo-client- --exclude 'logstash-mweibo-client-2015.05.11'
curator --timeout 36000 --host 10.0.0.100 delete indices --older
-than 30 --time-unit days --timestring '%Y.%m.%d' --regex '^logs
tash-mweibo-\d+'
curator --timeout 36000 --host 10.0.0.100 close indices --older-
than 7 --time-unit days --timestring '%Y.%m.%d' --prefix logstas
h-
curator --timeout 36000 --host 10.0.0.100 optimize --max_num_seg
ments 1 indices --older-than 1 --newer-than 7 --time-unit days -
-timestring '%Y.%m.%d' --prefix logstash-
```

这一顿任务，结果是：

logstash-mweibo-nginx-yyyy.mm.dd 索引保存最近 5 天，*logstash-mweibo-client-yyyy.mm.dd* 保存最近 10 天，*logstash-mweibo-yyyy.mm.dd* 索引保存最近 30 天；且所有七天前的 *logstash-** 索引都暂时关闭不用；最后对所有非当日日志做 segment 合并优化。

profiler

profiler 是 Elasticsearch 5.0 的一个新接口。通过这个功能，可以看到一个搜索聚合请求，是如何拆分成底层的 Lucene 请求，并且显示每部分的耗时情况。

启用 profiler 的方式很简单，直接在请求里加一行即可：

```
curl -XPOST 'http://localhost:9200/_search' -d '{
    "profile": true,
    "query": { ... },
    "aggs": { ... }
}'
```

可以看到其中对 query 和 aggs 部分的返回是不太一样的。

query

query 部分包括 collectors、rewrite 和 query 部分。对复杂 query，profiler 会拆分 query 成多个基础的 TermQuery，然后每个 TermQuery 再显示各自的分阶段耗时如下：

```
"breakdown": {
    "score": 51306,
    "score_count": 4,
    "build_scorer": 2935582,
    "build_scorer_count": 1,
    "match": 0,
    "match_count": 0,
    "create_weight": 919297,
    "create_weight_count": 1,
    "next_doc": 53876,
    "next_doc_count": 5,
    "advance": 0,
    "advance_count": 0
}
```

aggs

```
        "time": "1124.864392ms",
        "breakdown": {
            "reduce": 0,
            "reduce_count": 0,
            "build_aggregation": 1394,
            "build_aggregation_count": 150,
            "initialise": 2883,
            "initialize_count": 150,
            "collect": 1124860115,
            "collect_count": 900
        }
```

我们可以很明显的看到聚合统计在初始化阶段、收集阶段、构建阶段、汇总阶段分别花了多少时间，遍历了多少数据。

注意其中 *reduce* 阶段还没实现完毕，所有都是 *0*。因为目前 *profiler* 只能在 *shard* 级别上做统计。

collect 阶段的耗时，有助于我们调整对应 aggs 的 `collect_mode` 参数选择。目前 Elasticsearch 支持 `breadth_first` 和 `depth_first` 两种方式。

initialise 阶段的耗时，有助于我们调整对应 aggs 的 `execution_hint` 参数选择。目前 Elasticsearch 支持 `map`、`global_ordinals_low_cardinality`、`global_ordinals` 和 `global_ordinals_hash` 四种选择。在计算离散度比较大的字段统计值时，适当调整该参数，有益于节省内存和提高计算速度。

对高离散度字段值统计性能很关注的读者，可以关注 https://github.com/elastic/elasticsearch/pull/21626 这条记录的进展。

扩展和测试方案

在体验完 Elasticsearch 便捷的操作后，下一步一定会碰到的问题是：数据写入变慢了，机器变卡了，是需要做优化呢？还是需要扩容设备了？如果做扩容，索引的分片和副本设置多少才合适？如果做优化，某个参数能造成什么样的影响？

而 ES 集群性能，受服务器硬件、数据结构和长度、请求接口复杂度等各种环节影响颇大。这些问题，都需要有一个标准的测试流程给出答案。

由于 ES 是近乎线性扩展的分布式系统，所以对上述需求我们都可以总结成同一个测试模式：

1. 使用和线上集群相同硬件配置的服务器搭建一个单节点集群。
2. 使用和线上集群相同的映射创建一个 0 副本，1 分片的测试索引。
3. 使用和线上集群相同的数据写入进行压测。
4. 观察写入性能，或者运行查询请求观察搜索聚合性能。
5. 持续压测数小时，使用监控系统记录 eps、requesttime、fielddata cache、GC count 等关键数据。

测试完成后，根据监控系统数据，确定单分片的性能拐点，或者适合自己预期值的临界点。这个数据，就是一个基准数据。之后的扩容计划，都可以以这个基准单位进行。

需要注意的是，测试是以分片为单位的，在实际使用中，因为主分片和副本分片都是在各自节点做 indexing 和 merge 操作，需要消耗同样的写入性能。所以，实际集群的容量预估中，要考虑副本数的影响。也就是说，假如你在基准测试中得到单机写入性能在 10000 eps，那么开启一个副本后所能达到的 eps 就只有 5000 了。还想写入 10000 eps 的话，就需要加一倍机器。

另外，测试中我们使用的配置都尽量贴合当前现状。事实上，很多配置可能其实并不合理。在确定基准线并开始扩容之前，还是要认真调节配置，审核请求使用的接口是否最优，然后反复测试。然后取一个最终的基准值。

审核请求，更是一个长期的过程，就像 DBA 永远需要关注慢查询一样。ES 的慢查询请求处理，请阅读稍后性能日志一节。

esrally 测试工具

esrally 是 Elastic.co 开源的专门做 Elasticsearch 压测的工具。我们在官网上看到的 nightly benchmark 结果就是用这个工具每晚运行生成的报告。用这个工具，可以很方便的验证自己的代码修改、配置调整对性能的影响效果。

安装运行

esrally 依赖 python3.4+，所以需要先安装好 python 3.5。然后直接 `pip3 install esrally` 即可。

被压测的 Elasticsearch 有两种来源：

- 本机有 gradle 工具的，可以从最新的 GitHub master 代码编译
- 没有 gradle 工具的，可以按官方提供的标签，下载对应版本的二进制分发包。

esrally 在压测完毕后，可以把指标数据写入到另一个 ES 索引中，可以很方便的用 Kibana 做图表可视化。这就需要另外配置一下 `~/.rally/rally.ini` 里 reporting 部分的参数：

```
[reporting]
datastore.type = elasticsearch
datastore.host = localhost
datastore.port = 9200
datastore.secure = False
datastore.user =
datastore.password =
```

不用担心这个 ES 会跟一会儿压测运行的 ES 冲突，因为压测启动的 ES 会监听在其他端口上。

我们先简单测试一下标准的运行：：

```
/opt/local/Library/Frameworks/Python.framework/Versions/3.5/bin/
esrally --pipeline=from-distribution --distribution-version=5.0.
0
```

默认情况下压测采用的数据集叫 geonames，是一个 2.8GB 大的 JSON 数据。ES 也提供了一系列其他类型的压测数据集。如果要切换数据集采用 --track 参数：

```
/opt/local/Library/Frameworks/Python.framework/Versions/3.5/bin/
esrally --pipeline=from-distribution --distribution-version=5.0.
0 --track=geonames
```

重复运行的时候可以修改 `~/.rally/rally.ini` 里的 `tracks[default.url]` ，改为第一次运行时下载的地址：~/.rally/benchmarks/tracks/default 。然后采用离线参数重复运行：

```
/opt/local/Library/Frameworks/Python.framework/Versions/3.5/bin/
esrally --offline --pipeline=from-distribution --distribution-ve
rsion=5.0.0 --track=geonames
```

静静等待程序运行完毕，就会给出一个漂亮的输出结果了。

调整压测任务

默认一次压测运行会是一个很漫长的时间，如果你其实只关心部分的性能，比如只关心写入，不关心搜索。其实可以自己去修改一下 track 的任务定义。

track 的定义文件在 `~/.rally/benchmarks/tracks/default/geonames/track.json` 。如果你改动较大，建议直接新建一个 track 目录，比如叫 `mytest/track.json` 。

对照 geonames 里的定义，一个 track 包括以下部分：

- meta：定义数据来源 URL。
- indices：定义索引名称、索引 mapping 的文件位置、数据的存放位置和校验信息。
- operations：定义一个个操作的名称、类型、索引和请求参数。如果操作类型是 index，可用的索引参数有：client 并发量、bulk 大小、是否强制 merge 等；如果操作类型是 search，可用的请求参数就是一个 queries 数组，按序放好一个个 queryDSL。
- challenges：定义好名称和调用哪些 operation，调用顺序如何。

最后运行命令的时候通过 --challenge= 参数来指定执行哪个任务。

比如我们只关心写入不关心搜索，打开 track.json 可以看到有这么几个 challenges：

```
"challenges": [
    {
        "name": "append-no-conflicts",
        "description": "",
        "schedule": [
            "index-append-default-settings",
            "stats",
            "search"
        ]
    },
    {
        "name": "append-fast-no-conflicts",
        "description": "",
        "schedule": [
            "index-append-fast-settings"
        ]
    },
```

我们就知道了，默认的 `append-no-conflicts` 是要测完写入再测搜索的，而 `append-fast-no-conflicts` 是只测写入的。那么我们这么运行就行：

```
/opt/local/Library/Frameworks/Python.framework/Versions/3.5/bin/
esrally --offline --pipeline=from-distribution --distribution-ve
rsion=5.0.0 --track=geonames --challenge=append-fast-no-conflict
s
```

调整压测数据

如果要用自己的数据集呢，也一样是在自己的 track.json 里定义，比如：

```
{
    "meta": {
        "data-url": "/Users/raochenlin/.rally/benchmarks/data/splunklog/1468766825_10.json.bz2"
    },
    "indices": [
        {
            "name": "splunklog",
            "types": [
                {
                    "name": "type",
                    "mapping": "mappings.json",
                    "documents": "1468766825_10.json.bz2",
                    "document-count":  924645,
                    "compressed-bytes": 19149532,
                    "uncompressed-bytes": 938012996
                }
            ]
        }
    ],
```

这里就是用的一份 splunkd 的 internal 日志，JSON 导出。字节数大小为 938012996。压缩后为 19149532。

多集群连接

当你的 ES 集群发展到一定规模，单集群不足以应对庞大的在线索引量级，或者由于业务隔离需求，都有可能划分成多个集群。这时候，另一个问题就出来了：可能其中有一部分数据，被分割在两个集群里，但是还是需要一起使用的。如果是自己写程序，当然可以初始化两个对象，分别连接两个集群，得到结果集后再自行合并。但是如果用 Elastic Stack 的，Kibana 可不支持同时连接两个集群地址，这时候，就要用到 ES 中一个特殊的角色：tribe 节点。

tribe 节点只需要提供集群自动发现方面的配置，连接上多个集群后，对外提供只读功能。 `elasticsearch.yml` 配置示例如下：

```
tribe:
    1002:
        cluster.name: es1002
        discovery.zen.ping.timeout: 100s
        discovery.zen.ping.multicast.enabled: false
        discovery.zen.ping.unicast.hosts: ["10.19.0.22","10.19.0
.24",10.19.0.21"]
    1003:
        cluster.name: es1003
        discovery.zen.ping.timeout: 100s
        discovery.zen.ping.multicast.enabled: false
        discovery.zen.ping.unicast.hosts: ["10.19.0.97","10.19.0
.98","10.19.0.99","10.19.0.100"]
    blocks:
        write:    true
        metadata: true
    on_conflict: prefer_1003
```

注意这里的 `on_conflict` 设置，当多个集群内，索引名称有冲突的时候，tribe 节点默认会把请求轮询转发到各个集群上，这显然是不可以的。所以可以设置一个优先级，在索引名冲突的时候，偏向于转发给某一个集群。

以 tribe 配置启动的 Elasticsearch 服务，其日志输入如下：

```
[2015-06-18 18:05:51,983][INFO ][node                     ] [Man
```

```
slaughter] version[1.5.1], pid[12846], build[5e38401/2015-04-09T
13:41:35Z]
[2015-06-18 18:05:51,984][INFO ][node                     ] [Man
slaughter] initializing ...
[2015-06-18 18:05:51,990][INFO ][plugins                  ] [Man
slaughter] loaded [], sites []
[2015-06-18 18:05:54,891][INFO ][node                     ] [Man
slaughter/1003] version[1.5.1], pid[12846], build[5e38401/2015-0
4-09T13:41:35Z]
[2015-06-18 18:05:54,891][INFO ][node                     ] [Man
slaughter/1003] initializing ...
[2015-06-18 18:05:54,891][INFO ][plugins                  ] [Man
slaughter/1003] loaded [], sites []
[2015-06-18 18:05:55,654][INFO ][node                     ] [Man
slaughter/1003] initialized
[2015-06-18 18:05:55,655][INFO ][node                     ] [Man
slaughter/1002] version[1.5.1], pid[12846], build[5e38401/2015-0
4-09T13:41:35Z]
[2015-06-18 18:05:55,655][INFO ][node                     ] [Man
slaughter/1002] initializing ...
[2015-06-18 18:05:55,656][INFO ][plugins                  ] [Man
slaughter/1002] loaded [], sites []
[2015-06-18 18:05:56,275][INFO ][node                     ] [Man
slaughter/1002] initialized
[2015-06-18 18:05:56,285][INFO ][node                     ] [Man
slaughter] initialized
[2015-06-18 18:05:56,286][INFO ][node                     ] [Man
slaughter] starting ...
[2015-06-18 18:05:56,486][INFO ][transport                ] [Man
slaughter] bound_address {inet[/0:0:0:0:0:0:0:0:9301]}, publish_
address {inet[/10.19.0.100:9301]}
[2015-06-18 18:05:56,499][INFO ][discovery                ] [Man
slaughter] elasticsearch/Oewo-L2fR3y2xsgpsoI4Og
[2015-06-18 18:05:56,499][WARN ][discovery                ] [Man
slaughter] waited for 0s and no initial state was set by the dis
covery
[2015-06-18 18:05:56,529][INFO ][http                     ] [Man
slaughter] bound_address {inet[/0:0:0:0:0:0:0:0:9201]}, publish_
address {inet[/10.19.0.100:9201]}
[2015-06-18 18:05:56,530][INFO ][node                     ] [Man
```

```
slaughter/1003] starting ...
[2015-06-18 18:05:56,603][INFO ][transport                ] [Man
slaughter/1003] bound_address {inet[/0:0:0:0:0:0:0:0:9302]}, pub
lish_address {inet[/10.19.0.100:9302]}
[2015-06-18 18:05:56,609][INFO ][discovery                ] [Man
slaughter/1003] es1003/m1-cDaFTSoqqyC2iiQhECA
[2015-06-18 18:06:26,610][WARN ][discovery                ] [Man
slaughter/1003] waited for 30s and no initial state was set by t
he discovery
[2015-06-18 18:06:26,610][INFO ][node                     ] [Man
slaughter/1003] started
[2015-06-18 18:06:26,611][INFO ][node                     ] [Man
slaughter/1002] starting ...
[2015-06-18 18:06:26,674][INFO ][transport                ] [Man
slaughter/1002] bound_address {inet[/0:0:0:0:0:0:0:0:9303]}, pub
lish_address {inet[/10.19.0.100:9303]}
[2015-06-18 18:06:26,676][INFO ][discovery                ] [Man
slaughter/1002] es1002/4FPiRPh7TFyBk-BaPc_TLg
[2015-06-18 18:06:56,676][WARN ][discovery                ] [Man
slaughter/1002] waited for 30s and no initial state was set by t
he discovery
[2015-06-18 18:06:56,677][INFO ][node                     ] [Man
slaughter/1002] started
[2015-06-18 18:06:56,677][INFO ][node                     ] [Man
slaughter] started
[2015-06-18 18:07:37,266][INFO ][cluster.service          ] [Man
slaughter/1003] detected_master [10.19.0.97][jnA-rt2fS_22Mz9nYl5
Ueg][localhost.localdomain][inet[/10.19.0.97:9300]]{max_local_st
orage_nodes=1, data=false, master=true}, added {[10.19.0.73][_S8
ylz1OTv6Nyp1YoMRNGQ][esnode073.mweibo.bx.sinanode.com][inet[/10.
19.0.73:9300]]{max_local_storage_nodes=1, master=false},}, reaso
n: zen-disco-receive(from master [[10.19.0.97][jnA-rt2fS_22Mz9nY
l5Ueg][localhost.localdomain][inet[/10.19.0.97:9300]]{max_local_
storage_nodes=1, data=false, master=true}])
[2015-06-18 18:07:37,382][INFO ][tribe                    ] [Man
slaughter] [1003] adding node [[10.19.0.73][_S8ylz1OTv6Nyp1YoMRN
GQ][esnode073.mweibo.bx.sinanode.com][inet[/10.19.0.73:9300]]{ma
x_local_storage_nodes=1, tribe.name=1003, master=false}]
[2015-06-18 18:07:37,391][INFO ][tribe                    ] [Man
slaughter] [1003] adding node [[Manslaughter/1003][m1-cDaFTSoqqy
```

```
C2iiQhECA][localhost.localdomain][inet[/10.19.0.100:9302]]{data=
false, tribe.name=1003, client=true}]
[2015-06-18 18:07:37,393][INFO ][tribe                    ] [Man
slaughter] [1003] adding node [[10.19.0.97][_mIrWKzZTYifp1xshngB
ew][esnode054.mweibo.bx.sinanode.com][inet[/10.19.0.54:9300]]{ma
x_local_storage_nodes=1, tribe.name=1003, master=false}]
[2015-06-18 18:07:37,393][INFO ][tribe                    ] [Man
slaughter] [1003] adding index [logstash-mweibo-vip-2015.06.15]
[2015-06-18 18:07:37,394][INFO ][tribe                    ] [Man
slaughter] [1003] adding index [logstash-php-2015.06.08]
[2015-06-18 18:07:37,394][INFO ][tribe                    ] [Man
slaughter] [1003] adding index [logstash-mweibo-vip-2015.06.16]
[2015-06-18 18:07:37,395][INFO ][tribe                    ] [Man
slaughter] [1003] adding index [.kibana]
[2015-06-18 18:07:37,398][INFO ][tribe                    ] [Man
slaughter] [1003] adding index [logstash-php-2015.06.14]
[2015-06-18 18:07:37,403][INFO ][tribe                    ] [Man
slaughter] [1003] adding index [logstash-mweibo-vip-2015.06.10]
[2015-06-18 18:07:37,403][INFO ][tribe                    ] [Man
slaughter] [1003] adding index [kibana-int]
[2015-06-18 18:07:37,404][INFO ][tribe                    ] [Man
slaughter] [1003] adding index [logstash-mweibo-2015.06.13]
[2015-06-18 18:07:37,411][INFO ][cluster.service          ] [Man
slaughter] added {[10.19.0.73][_S8ylz1OTv6Nyp1YoMRNGQ][esnode073
.mweibo.bx.sinanode.com][inet[/10.19.0.73:9300]]{max_local_stora
ge_nodes=1, tribe.name=1003, master=false},[10.19.0.97][jnA-rt2f
S_22Mz9nYl5Ueg][localhost.localdomain][inet[/10.19.0.97:9300]]{m
ax_local_storage_nodes=1, tribe.name=1003, data=false, master=tr
ue},}, reason: cluster event from 1003, zen-disco-receive(from m
aster [[10.19.0.97][jnA-rt2fS_22Mz9nYl5Ueg][localhost.localdomai
n][inet[/10.19.0.97:9300]]{max_local_storage_nodes=1, data=false
, master=true}])
[2015-06-18 18:08:07,316][INFO ][cluster.service          ] [Man
slaughter/1002] detected_master [10.19.0.22][6qyQh9EURUyO7RBC_dX
Dow][localhost.localdomain][inet[/10.19.0.22:9300]]{max_local_st
orage_nodes=1, master=true}, added {[10.19.0.93][qAklY08iSsSfIf2
vvu6Iyw][localhost.localdomain][inet[/10.19.0.93:9300]]{max_loca
l_storage_nodes=1, master=false}])
[2015-06-18 18:08:07,350][INFO ][indices.breaker          ] [Man
slaughter/1002] Updating settings parent: [PARENT,type=PARENT,li
```

```
mit=259489792/247.4mb,overhead=1.0], fielddata: [FIELDDATA,type=
MEMORY,limit=155693875/148.4mb,overhead=1.03], request: [REQUEST
,type=MEMORY,limit=103795916/98.9mb,overhead=1.0]
[2015-06-18 18:08:07,353][INFO ][tribe                    ] [Man
slaughter] [1002] adding node [[10.19.0.93][qAklY08iSsSfIf2vvu6I
yw][localhost.localdomain][inet[/10.19.0.93:9300]]{max_local_sto
rage_nodes=1, tribe.name=1002, master=false}]
[2015-06-18 18:08:07,357][INFO ][tribe                    ] [Man
slaughter] [1002] adding node [[Manslaughter/1002][4FPiRPh7TFyBk
-BaPc_TLg][localhost.localdomain][inet[/10.19.0.100:9303]]{data=
false, tribe.name=1002, client=true}]
[2015-06-18 18:08:07,358][INFO ][tribe                    ] [Man
slaughter] [1002] adding node [[10.19.0.22][tkrBsbnLTry0zzZEdbQR
0A][localhost.localdomain][inet[/10.19.0.27:9300]]{max_local_sto
rage_nodes=1, tribe.name=1002, master=false}]
[2015-06-18 18:08:07,358][INFO ][tribe                    ] [Man
slaughter] [1002] adding index [test.yingju1-mweibo_client_downs
tream_success-2015.06.07]
[2015-06-18 18:08:07,363][INFO ][tribe                    ] [Man
slaughter] [1002] adding index [logstash-mweibo_client_downstrea
m_error-2015.06.02]
[2015-06-18 18:08:07,366][INFO ][tribe                    ] [Man
slaughter] [1002] adding index [.kibana_5601]
[2015-06-18 18:08:07,377][INFO ][cluster.service          ] [Man
slaughter] added {[10.19.0.22][6qyQh9EURUyO7RBC_dXDow][localhost
.localdomain][inet[/10.19.0.22:9300]]{max_local_storage_nodes=1,
 tribe.name=1002, master=false},[10.19.0.93][l7nkk-H7S6GvMzWwGe0
_CA][localhost.localdomain][inet[/10.19.0.93:9300]]{max_local_st
orage_nodes=1, tribe.name=1002, master=false},}, reason: cluster
 event from 1002, zen-disco-receive(from master [[10.19.0.22][6q
yQh9EURUyO7RBC_dXDow][localhost.localdomain][inet[/10.19.0.22:93
00]]{max_local_storage_nodes=1, master=true}])
[2015-06-18 18:08:13,208][DEBUG][discovery.zen.publish    ] [Man
slaughter/1003] received cluster state version 782404
[2015-06-18 18:08:21,803][DEBUG][discovery.zen.publish    ] [Man
slaughter/1003] received cluster state version 782405
[2015-06-18 18:08:33,229][DEBUG][discovery.zen.publish    ] [Man
slaughter/1003] received cluster state version 782406
```

日志中可以明显看到，节点是如何分别连接上两个集群的。

最后，我们可以使用标准的 RESTful 接口来验证一下：

```
# curl 10.19.0.100:9201/_cat/indices?v
health status index
       pri rep docs.count docs.deleted store.size pri.store.size
green  open   test.yingju1-mweibo_client_downstream_success-2015
.06.07  20   1   40692459            0    154.1gb           77gb
green  open   weibo-client-video-2015.06.19
         5   1          0            0      970b           575b
green  open   dpool-pc-weibo-2015.06.19
        20   1          0            0     3.7kb          2.2kb
green  open   logstash-video-2015.06.16
        27   0  149015413            0    13.4gb         13.4gb
```

不同集群的索引，都可以通过 tribe node 访问到了。

alias的几点应用

本节作者：*childe*

索引更改名字时, 无缝过渡

情景1

用Logstash采集当前的所有nginx日志, 放入ES, 索引名叫nginx-YYYY.MM.DD.

后来又增加了apache日志, 希望能放在同一个索引里面,统一叫web-YYYY.MM.DD.

我们只要把Logstash配置更改一下,然后重启, 数据就会写入新的索引名字下. 但是同一天的索引就会被分成了2个, kibana上面就不好配置了.

如此实现

1. 今天是2015.07.28. 我们为nginx-2015.07.28建一个alias叫做web-2015.07.28, 之前的所有nginx日志也如此照做.
2. kibana中把dashboard配置的索引名改成web-YYYY.MM.DD
3. 将logstash里面的elasticsearch的配置改成web-YYYY.MM.DD, 重启.
4. 无缝切换实现.

情景2

用Logstash采集当前的所有nginx日志, 放入ES, 索引名叫nginx-YYYY.MM.DD.

某天(2015.07.28)希望能够按月建立索引, 索引名改成nginx-YYYY.MM.

[注意] 像情景1中那样新建一个叫nginx-2015.07的alias, 并指向本月的其他的索引是不行的. 因为一个alias指向了多个索引, 写这个alias的时候, ES不可能知道写入哪个真正的索引.

如此实现

1. 新建索引nginx-2015.07, 以及他的alias: nginx-2015.07.29, nginx-2015.07.30 ... 等. 新建索引nginx-2015.08, 以及他的alias: nginx-2015.08.01, nginx-2015.08.02 ... 等.
2. 等到第二天, 将logstash配置更改为nginx-YYYY.MM, 重启.
3. 如果索引只保留10天(一般来说, 不可能永久保存), 在10天之后的某天, 将kibana配置更改为 `[nginx-]YYYY.MM`

缺点

第二步, 第三步需要记得手工操作, 或者写一个crontab定时任务.

另外一种思路

1. 新建一个叫nginx-2015.07的alias, 并指向本月的其他的索引. 这个时候不能马上更改logstash配置写入nginx-2015.07. 因为一个alias指向了多个索引, 写这个alias的时候, ES不可能知道写入哪个真正的索引.
2. 新建索引nginx-2015.08, 以及他的alias: nginx-2015.08.01, nginx-2015.08.02 ... 等.
3. 把kibana配置改为 `[nginx-]YYYY.MM` .
4. 到2015.08.01这天, 或者之后的某天, 更改logstash配置, 把elasticsearch的配置更改为nginx-YYYY.MM. 重启.

缺点

1. 7月份的索引还是按天建立.
2. 第四步需要记得手工操作.

情景2的操作有些麻烦, 这些都是为了"无缝"这个大前提. 我们不希望用户(kibana的使用者)感觉到任何变化, 更不能让使用者感受到数据有缺失.

如果用户愿意妥协, 那ELK的管理者就可以马上简单粗暴的把数据写到nginx-2015.07, 并把kibana配置改过去. 这样一来, 用户可能就不能方便的看到之前的数据了.

按域名配置kibana的dashboard

nginx日志中有个字段是domain, 各个业务部门在Kibana中需要看且只看自己域名下的日志.

可以在logstash中按域名切分索引,但ES的metadata会成倍的增长,带来极大的负担.

也可以放在nginx-YYYY.MM.DD一个索引中, 在kibana加一个filter过滤. 但这个关键的filter和其它所有filter排在一起,是很容易很不小心清掉的.

我们可以在template中配置, 对每一个域名做一个alias.

```
"aliases" : {
      "{index}-www.corp.com" : {
        "filter" : {
          "term" : {
            "domain" : "www.corp.com"
          }
        }
      },
      "{index}-abc.corp.com" : {
        "filter" : {
          "term" : {
            "domain" : "abc.corp.com"
          }
        }
      }
}
```

当新的索引nginx-2015.07.28生成的时候, 会有nginx-2015.07.28-www.corp.com和nginx-2015.07.28-abc.corp.com等alias指向nginx-2015.07.28.

在kibana配置dashboard的时候, 就可以直接用alias做配置了.

在实际使用中, 可能还需要一个crontab定时的查询是否有新的域名加入, 自动对新域名做当天的alias, 并把它加入template.

映射与模板的定制

Elasticsearch 是一个 schema-less 的系统，但 schema-less 并不代表 no schema，而是 ES 会尽量根据 JSON 源数据的基础类型猜测你想要的字段类型映射。如果你对这种动态生成的映射关系不满意，或者想要使用一些更高级的映射设置，那么就需要使用自定义映射。

创建和更新映射

正如上面所说，ES 可以随时根据数据中的新字段来创建新的映射关系。所以，我们也可以自己在还没有正式数据写入之前，先创建一个基础的映射。等后续数据有其他字段时，ES 也一样会自动处理。

映射的的创建方式如下：

```
# curl -XPUT http://127.0.0.1:9200/logstash-2015.06.20/_mapping
-d '
{
  "mappings": {
    "syslog" : {
      "properties" : {
        "@timestamp" : {
          "type" : "date"
        },
        "message" : {
          "type" : "text"
        },
        "pid" : {
          "type" : "long"
        }
      }
    }
  }
}'
```

注意：对于已存在的映射，ES 的自动处理仅限于新字段出现。已经生成的字段映射，是不可变更的。如果确实需要，请参阅之前的 reindex 接口小节，采用重新导入数据的方式完成。

而如果是新增一个字段映射的更新，那还是可以通过 `/_mapping` 接口直接完成的：

```
# curl -XPUT http://127.0.0.1:9200/logstash-2015.06.21/_mapping/
syslog -d '
{
  "properties" : {
    "syslogtag" : {
      "type" : "keyword",
    }
  }
}'
```

没错，这里只需要单独写这个新字段的内容就够了。ES 会自动合并进去。

删除映射

删除数据并不代表会删除数据的映射。比如：

```
# curl -XDELETE http://127.0.0.1:9200/logstash-2015.06.21/syslog
```

删除了索引下 syslog 的全部数据，但是 syslog 的映射还在。删除映射(同时也就删掉了数据)的命令是：

```
# curl -XDELETE http://127.0.0.1:9200/logstash-2015.06.21/_mappi
ng/syslog
```

当然，如果删除整个索引，那映射也是同时被清除的。

核心类型

mapping 中主要就是针对字段设置类型以及类型相关参数。那么，我们首先来了解一下 Elasticsearch 支持的核心类型：

1. JSON 基础类型
2. 字符串: text, keyword
3. 数字: byte, short, integer, long, float, double，half_float
4. 时间: date
5. 布尔值: true, false
6. 数组: array
7. 对象: object
8. ES 独有类型
9. 多重: multi
10. 经纬度: geo_point
11. 网络地址: ip
12. 堆叠对象: nested object
13. 二进制: binary
14. 附件: attachment

前面提到，ES 是根据收到的 JSON 数据里的类型来猜测的。所以，一个内容为 `"123"` 的数据，猜测出来的类型应该是字符串而不是数值。除非这个字段已经有了确定为 long 的映射关系，那么 ES 会尝试做一次转换。如果转换失败，这条数据写入就会报错。

查看已有数据的映射

学习索引映射最直接的方式，就是查看已有数据索引的映射。我们用 logstash 写入 ES 的数据，都会根据 logstash 自带的 template，生成一个很有学习意义的映射：

```
# curl -XGET http://127.0.0.1:9200/logstash-2015.06.16/_mapping/
tweet
{
   "gb": {
      "mappings": {
         "tweet": {
            "properties": {
               "date": {
                  "type": "date",
                  "format": "dateOptionalTime"
               },
               "name": {
                  "type": "keyword"
               },
               "tweet": {
                  "type": "text"
               },
               "user_id": {
                  "type": "long"
               }
            }
         }
      }
   }
}
```

自定义字段映射

大家可以通过上面一个现存的映射发现其实所有的字段都有好几个属性，这些都是我们可以自己定义修改的。除了已经看到的这些基本内容外，ES 还支持其他一些可能会比较常用的映射属性：

- 索引还是存储
- 自定义分词器
- 自定义日期格式

精确索引

字段都有几个基本的映射选项，类型(type)、存储(store)和索引方式(index)。默认来说，store 是 false 而 index 是 true。因为 ES 会直接在 `_source` 里存储全部 JSON，不用每个 field 单独存储了。

不过在非日志场景，比如用作监控存储的 TSDB 使用的时候，我们就可以关闭 `_source`，只存储有关 metric 名称的字段 store；同时也关闭所有数值字段的 `index`，只使用它们的 `doc_values`。

时间格式

稍微见过 Elastic Stack 示例的人，都对其中 `@timestamp` 字段的特殊格式有深刻的印象。这个时间格式在 Nginx 中叫 `$time_iso8601`，在 Rsyslog 中叫 `date-rfc3339`，在 ES 中叫 `dateOptionalTime`。但事实上，ES 完全可以接收其他时间格式作为时间字段的内容。对于 ES 来说，时间字段内容实际都是转换成 long 类型作为内部存储的。所以，接收段的时间格式，可以任意配置：

```
"@timestamp" : {
    "type" : "date"
    "format" : "dd/MMM/YYYY:HH:mm:ss Z",
}
```

而 ES 默认的时间字段格式，除了 `dateOptionalTime` 以外，还有一种，就是 `epoch_millis`，毫秒级的 UNIX 时间戳。因为这个数值 ES 可以直接毫不修改的存成内部实际的 long 数值。此外，从 ES 2.0 开始，新增了对秒级 UNIX 时间戳的支持，其 `format` 定义为：`epoch_second`。

注意：从 ES 2.x 开始，同名 date 字段的 `format` 也必须保持一致。

多重索引

多重索引是 logstash 用户最习惯的一个映射，因为这是 logstash 默认设置开启的配置：

```
"title": {
    "type": "text",
    "fields": {
        "raw": { "type": "keyword" }
    }
}
```

其作用是，在 *title* 字段数据写入的时候，ES 会自动生成两个字段，分别是 `title` 和 `title.raw` 。这样，在可能同时需要分词与不分词结果的环境下，就可以很灵活的使用不同的索引字段了。比如，查看标题中最常用的单词，应该使用 `title` 字段；查看阅读数最多的文章标题，应该使用 `title.raw` 字段。

注意：raw 这个名字你可以自己随意取。比如说，如果你绝大多数时候用的是精确索引，那么你完全可以为了方便反过来定义：

```
"title": {
    "type": "keyword",
    "fields": {
        "alz": { "type": "text" }
    }
}
```

特殊字段

上面介绍的，都是对普通数据字段的一些常用设置。而实际上，ES 默认还有一些特殊字段，在默默的发挥着作用。这些字段，统一以 `_` 下划线开头。在之前 CRUD 章节中，我们就已经看到一些，比如 `_index` ， `_type` ， `_id` 。默认不开启的还有 `_parent` 等。这里需要介绍三个，对我们索引和检索的效果和性能，都有较大影响的：

_all

`_all` 里存储了各字段的数据内容。其作用是，在检索的时候，如果无法或者未指明具体搜索哪个字段的数据，那么 ES 默认就会是从 `_all` 里去查找。

对于日志场景，如果你的日志划分出来的字段比较少且数目固定。那么，完全可以关闭掉 `_all` 功能，节省这部分 IO 和 CPU。

```
"_all" : {
    "enabled" : false
}
```

Elastic.co 甚至考虑在 6.0 版本中废弃掉 `_all`，由用户自定义字段来完成类似工作(比如日志场景中的 `message` 字段)。因为 `_all` 采用的分词器和用户自定义字段可能是不一致的，某些场景下会产生误解。

_field_names

`_field_names` 里存储的是每条数据里的字段名，你可以认为它是 `_all` 的补集。其主要作用是在做 `_missing_` 或 `_exists_` 查询的时候，不用检索数据本身，直接获取字段名对应的文档 ID。听起来似乎蛮不错的，但是文档较多的时候，就意味着这个倒排链非常长！而且几乎每次索引写入操作，都需要往这个倒排里加入文档 ID，这点是实际使用中非常损耗写入性能的地方。

除非有必要理由，关闭 `_field_names` 可以提升大概 20% 的写入性能。

_source

`_source` 里存储了该条记录的 JSON 源数据内容。这部分内容只是按照 ES 接收到的内容原样存储下来，并不经过索引过程。对于 ES 的请求过程来说，它不参与 Query 阶段，而只用于 Fetch 阶段。我们在 GET 或者 `/_search` 时看到的数据内容，都是从 `_source` 里获取到的。

所以，虽然 `_source` 也重复了一遍索引中的数据，一般我们并不建议关闭这个功能。因为一旦关闭，你搜索的结果除了一个 `_id`，啥都看不到。对于日志场景，意义不是很大。

当然，也有少数场景是可以关闭 `_source` 的：

1. 把 ES 作为时间序列数据库使用，只要聚合统计结果，不要源数据内容。
2. 把 ES 作为纯检索工具使用，`_id` 对应的内容在 HDFS 上另外存储，搜索后使用所得 `_id` 去 HDFS 上读取内容。

动态模板映射

不想使用默认识别的结果，单独设置一个字段的映射的方法，上面已经介绍完毕。那么，如果你有一类相似的数据字段，想要统一设置其映射，就可以用到下一项功能：动态模板映射(`dynamic_templates`)。

```
"_default_" : {
  "dynamic_templates" : [ {
    "message_field" : {
      "mapping" : {
        "omit_norms" : true,
        "store" : false,
        "type" : "text"
      },
      "match" : "*msg",
      "match_mapping_type" : "string"
    }
  }, {
    "string_fields" : {
      "mapping" : {
        "ignore_above" : 256,
        "store" : false,
        "type" : "keyword"
      },
      "match" : "*",
      "match_mapping_type" : "string"
    }
  } ],
  "properties" : {
  }
}
```

这样，只要字符串类型字段名以 msg 结尾的，都会经过全文索引，其他字符串字段则进行精确索引。同理，还可以继续书写其他类型(`long` , `float` , `date` 等)的 `match_mapping_type` 和 `match` 。

索引模板

对每个希望自定义映射的索引，都要定时提前通过发送 PUT 请求的方式创建索引的话，未免太过麻烦。ES 对此设计了索引模板功能。我们可以针对同一类索引，定制相同的模板。

模板中的内容包括两大类，setting(设置)和 mapping(映射)。setting 部分，多为在 `elasticsearch.yml` 中可以设置全局配置的部分，而 mapping 部分，则是这节之前介绍的内容。

如下为定义所有以 te 开头的索引的模板：

```
# curl -XPUT http://localhost:9200/_template/template_1 -d '
{
    "template" : "te*",
    "settings" : {
        "number_of_shards" : 1
    },
    "mappings" : {
        "type1" : {
            "_source" : { "enabled" : false }
        }
    }
}'
```

同时，索引模板是有序合并的。如果我们在同一类索引里，又想单独修改某一小类索引的一两处单独设置，可以再累加一层模板：

```
# curl -XPUT http://localhost:9200/_template/template_2 -d '
{
    "order" : 1,
    "template" : "tete*",
    "settings" : {
        "number_of_shards" : 2
    },
    "mappings" : {
        "type1" : {
            "_all" : { "enabled" : false }
        }
    }
}'
```

默认的 order 是 0，那么新创建的 order 为 1 的 *template_2* 在合并时优先级大于 *template_1*。最终，对 `tete*/type1` 的索引模板效果相当于：

```
{
    "settings" : {
        "number_of_shards" : 2
    },
    "mappings" : {
        "type1" : {
            "_source" : { "enabled" : false },
            "_all" : { "enabled" : false }
        }
    }
}
```

Puppet 自动部署 Elasticsearch

Elasticsearch 作为一个 Java 应用，本身的部署已经非常简单了。不过作为生产环境，还是有必要采用一些更标准化的方式进行集群的管理。Elasticsearch 官方提供并推荐使用 Puppet 方式部署和管理。其 Puppet 模块源码地址见：

https://github.com/elastic/puppet-elasticsearch

安装方法

和其他标准 Puppet Module 一样，puppet-elasticsearch 也可以通过 Puppet Forge 直接安装：

```
# puppet module install elasticsearch-elasticsearch
```

配置示例

安装好 Puppet 模块后，就可以使用了。模块提供几种 Puppet 资源，主要用法如下：

```
class { 'elasticsearch':
  version => '2.4.1',
  config => { 'cluster.name' => 'es1003' },
  java_install => true,
}
elasticsearch::instance { $fqdn:
  config => { 'node.name' => $fqdn },
  init_defaults => { 'ES_USER' => 'elasticsearch', 'ES_HEAP_SIZE
' => $memorysize > 64 ? '31g' : $memorysize / 2 },
  datadir => [ '/data1/elasticsearch' ],
}
elasticsearch::template { 'templatename':
  host => $::ipaddress,
  port => 9200,
  content => '{"template":"*","settings":{"number_of_replicas":0
}}'
}
```

示例中展示了三种资源：

- class: 配置具体安装的 Elasticsearch 软件版本，全集群公用的一些基础配置项。注意，puppet-elasticsearch 模块默认并不负责 Java 的安装，它只是调用操作系统对应的 Yum，Apt 工具，而 elasticsearch.rpm 或者 elasticsearch.deb 本身没有定义其他依赖(因为 java 版本太多了，定义起来不方便)。所以，如果依然要走 puppet-elasticsearch 配置来安装 Java 的话，需要额外开启 `java_install` 选项。该选项会使用另一个 Puppet Module —— puppetlabs-java 来安装，默认安装的是 jdk。如果你要节省空间，可以再加一行 `java_package` 来明确指定软件全名。
- instance: 配置具体单个进程实例的配置。其中 `config` 和 `init_defaults` 选项在 class 和 instance 资源中都可以定义，实际运行时，会自动做一次合并，当然，instance 里的配置优先级高于 class 中的配置。此外，最重要的定义是数据目录的位置。有多快磁盘的，可以在这里定义一个数组。
- template: 模板是 Elasticsearch 创建索引映射和设置时的预定义方式。一般可以通过在 `config/templates/` 目录下放置 JSON 文件，或者通过 RESTful API 上传配置两种方式管理。而这里，单独提供了 template 资源，通过 puppet 来管理模板。`content` 选项中直接填入模板内容，或者使用 `file` 选项读取文件均可。

事实上，模块还提供了 plugin 和 script 资源管理这两方面的内容。考虑在 ELK 中，二者用的不是很多，本节就不单独介绍了。想了解的读者可以参考官方文档。

集群版本升级

Elasticsearch 作为一个新兴项目，版本更新非常快。而且每次版本更新都或多或少带有一些重要的性能优化、稳定性提升等特性。可以说，ES 集群的版本升级，是目前 ES 运维必然要做的一项工作。

按照 ES 官方设计，有 restart upgrade 和 rolling upgrade 两种可选的升级方式。对于 1.0 版本以上的用户，推荐采用 rolling upgreade 方式。

但是，对于主要负载是数据写入的 **Elastic Stack** 场景来说，却并不是这样！

rolling upgrade 的步骤大致如下：

1. 暂停分片分配；
2. 单节点下线升级重启；
3. 开启分片分配；
4. 等待集群状态变绿后继续上述步骤。

实际运行中，步骤 2 的 ES 单节点从 restart 到加入集群，大概要 100s 左右的时间。也就是说，这 100s 内，该节点上的所有分片都是 unassigned 状态。而按照 Elasticsearch 的设计，数据写入需要至少达到 `replica/2+1` 个分片完成才能算完成。也就意味着你所有索引都必须至少有 1 个以上副本分片开启。

但事实上，很多日志场景，由于写入性能上的要求要高于数据可靠性的要求，大家普遍减小了副本数量，甚至直接关掉副本复制。这样一来，整个 rolling upgrade 期间，数据写入就会受到严重影响，完全丧失了 rolling 的必要性。

其次，步骤 3 中的 ES 分片均衡过程中，由于 ES 的副本分片数据都需要从主分片走网络复制重新传输一次，而由于重启，新升级的节点上的分片肯定全是副本分片(除非压根没副本)。在数据量较大的情况下，这个步骤耗时可能是几十分钟甚至以小时计。而且并发和限速上稍微不注意，可能导致分片均衡的带宽直接占满网卡，正常写入也还是受到影响。

所以，对于写入压力较大，数据可靠性要求偏低的实时日志场景，依然建议大家进行主动停机式的 restart upgrade。

restart upgrade 的步骤如下：

1. 首先适当加大集群的数据恢复和分片均衡并发度以及磁盘限速：

```
# curl -XPUT http://127.0.0.1:9200/_cluster/settings -d '{
  "persistent" : {
    "cluster" : {
      "routing" : {
        "allocation" : {
          "disable_allocation" : "false",
          "cluster_concurrent_rebalance" : "5",
          "node_concurrent_recoveries" : "5",
          "enable" : "all"
        }
      }
    },
    "indices" : {
      "recovery" : {
        "concurrent_streams" : "30",
        "max_bytes_per_sec" : "2gb"
      }
    }
  },
  "transient" : {
    "cluster" : {
      "routing" : {
        "allocation" : {
          "enable" : "all"
        }
      }
    }
  }
}'
```

1. 暂停分片分配：

```
# curl -XPUT http://127.0.0.1:9200/_cluster/settings -d '{
  "transient" : {
    "cluster.routing.allocation.enable" : "none"
  }
}'
```

1. 通过配置管理工具下发新版本软件包。
2. 公告周知后，停止数据写入进程(即 logstash indexer 等)
3. 如果使用 Elasticsearch 1.6 版本以上，可以手动运行一次 synced flush，同步副本分片的 commit id，缩小恢复时的网络传输带宽：

```
# curl -XPOST http://127.0.0.1:9200/_flush/synced
```

1. 全集群统一停止进程，更新软件包，重新启动。
2. 等待各节点都加入到集群以后，恢复分片分配：

```
# curl -XPUT http://127.0.0.1:9200/_cluster/settings -d '{
  "transient" : {
    "cluster.routing.allocation.enable" : "all"
  }
}'
```

由于同时启停，主分片几乎可以同时本地恢复，整个集群从 red 变成 yellow 只需要 2 分钟左右。而后的副本分片，如果有 synced flush，同样本地恢复，否则网络恢复总耗时，视数据大小而定，会明显大于单节点恢复的耗时。

1. 如果有 synced flush，建议等待集群变成 green 状态后，恢复写入；否则在集群变成 yellow 状态之后，即可着手开始恢复数据写入进程。

镜像备份

本节作者：李宏旭

大多数公司在使用 Elasticsearch 之前，都已经维护有一套 Hadoop 系统。因此，在实时数据慢慢变得冷却，不再被经常使用的时候，一个需求自然而然的就出现了：怎么把 Elasticsearch 索引数据快速转移到 HDFS 上，以解决 Elasticsearch 上的磁盘空间；而在我们需要的时候，又可以较快的从 HDFS 上把索引恢复回来继续使用呢？

Elasticsearch 为此提供了 snapshot 接口。通过这个接口，我们可以快速导入导出索引镜像到本地磁盘，网络磁盘，当然也包括 HDFS。

HDFS 插件安装配置

下载repository-hdfs插件，通过标准的 elasticsearch plugin 安装命令安装：

```
bin/plugin install elasticsearch/elasticsearch-repository-hdfs/2
.2.0
```

然后在 elasticsearch.yml 中增加以下配置：

```
# repository 配置
hdfs:
    uri:"hdfs://<host>:<port>"(默认port为8020)
    #Hadoop file-system URI
    path:"some/path"
    #path with the file-system where data is stored/loaded
    conf.hdfs_config:"/hadoop/hadoop-2.5.2/etc/hadoop/hdfs-site.
xml"
    conf.hadoop_config:"/hadoop/hadoop-2.5.2/etc/hadoop/core-sit
e.xml"
    load_defaults:"true"
    #whether to load the default Hadoop configuration (default)
or not
    compress:"false"
    # optional - whether to compress the metadata or not (defaul
t)
    chunk_size:"10mb"
    # optional - chunk size (disabled by default)
# 禁用 jsm
security.manager.enabled: false
```

默认情况下，Elasticsearch 为了安全考虑会在运行 JVM 的时候执行 JSM。出于 Hadoop 和 HDFS 客户端权限问题，所以需要禁用 JSM。将 `elasticsearch.yml` 中的 `security.manager.enabled` 设置为 `false` 。

将插件安装好，配置修改完毕后，需要重启 Elasticsearch 服务。没有重启节点插件可能会执行失败。

注意：**Elasticsearch** 集群的每个节点都要执行以上步骤！

Hadoop 配置

本节内容基于Hadoop版本：2.5.2，假定其配置文件目录：hadoop-2.5.2/etc/Hadoop。注意，安装hadoop集群需要建立主机互信，互信方法请自行查询，很简单。

相关配置文件如下：

mapred-site.xml.template

默认没有 mapred-site.xml 文件，复制 mapred-site.xml.template 一份，并把名字改为 mapred-site.xml，需要修改 3 处的 IP 为本机地址:

```
<configuration>
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
    <property>
        <name>mapreduce.jobtracker.http.address</name>
        <value>XX.XX.XX.XX:50030</value>
    </property>
    <property>
        <name>mapreduce.jobhistory.address</name>
        <value> XX.XX.XX.XX:10020</value>
    </property>
    <property>
        <name>mapreduce.jobhistory.webapp.address</name>
        <value> XX.XX.XX.XX:19888</value>
    </property>
</configuration>
```

yarn-site.xml

需要修改5处的IP为本机地址:

```
<configuration>

<!-- Site specific YARN configuration properties -->
    <property>
        <name>yarn.nodemanager.aux-services</name>
        <value>mapreduce_shuffle</value>
    </property>
        <property>
        <name>yarn.resourcemanager.address</name>
        <value> XX.XX.XX.XX:8032</value>
    </property>
    <property>
        <name>yarn.resourcemanager.scheduler.address</name>
        <value> XX.XX.XX.XX:8030</value>
    </property>
    <property>
        <name>yarn.resourcemanager.resource-tracker.address</nam
e>
        <value> XX.XX.XX.XX:8031</value>
    </property>
    <property>
        <name>yarn.resourcemanager.admin.address</name>
        <value> XX.XX.XX.XX:8033</value>
    </property>
    <property>
        <name>yarn.resourcemanager.webapp.address</name>
        <value> XX.XX.XX.XX:8088</value>
    </property>
</configuration>
```

hadoop-env.sh

修改 jdk 路径和 jvm 内存配置，内存使用根据情况配置。

```
export JAVA_HOME=/usr/java/jdk1.7.0_79
export HADOOP_PORTMAP_OPTS="-Xmx512m $HADOOP_PORTMAP_OPTS"
export HADOOP_CLIENT_OPTS="-Xmx512m $HADOOP_CLIENT_OPTS"
```

core-site.xml

临时目录及 hdfs 机器 IP 端口指定:

hadoop.tmp.dir /soft/hadoop-2.5.2/tmp Abase for other temporary directories. fs.defaultFS hdfs:// XX.XX.XX.XX:9000 io.file.buffer.size 4096

slaves

配置集群 IP 地址，集群有几个 IP 都要配置进去:

```
192.168.0.2
192.168.0.3
192.168.0.4
```

hdfs-site.xml

namenode 和 datenode 数据存放路径及别名:

```
<configuration>
    <property>
        <name>dfs.namenode.name.dir</name>
        <value>/data01/hadoop/name</value>
    </property>
    <property>
        <name>dfs.datanode.data.dir</name>
        <value>/data01/hadoop/data</value>
    </property>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
</configuration>
```

启动 Hadoop

格式化完成后也可使用 `sbin/start-all.sh` 启动，但有可能出现异常，建议按照顺序分开启动。

1. 首先需要格式化存储： `bin/Hadoop namenode –format`
2. 启动start-dfs.sh `sbin/start-dfs.sh`
3. 启动start-yarn.sh `sbin/start-yarn.sh`

备份导出

创建快照仓库

```
curl -XPUT 'localhost:9200/_snapshot/backup' -d
'{
    "type":"hdfs",
    "settings":{
        "path":"/test/repo",
        "uri":"hdfs://<uri>:<port>"
    }
}'
```

在这步可能会报错。通常是因为 hadoop 配置问题，更改好配置需要重新格式化文件系统:

在 hadoop 目录下执行 `bin/hadoop namenode -format`

索引快照

执行索引快照命令，可写入crontab，定时执行

```
curl -XPUT 'http://localhost:9200/_snapshot/backup/snapshot_1' -
d
 '{"indices":"indices_01,indices_02"}'
```

备份恢复

`curl -XPOST "localhost:9200/_snapshot/backup/snapshot_1/_restore"`

备份删除

```
curl -XDELETE "localhost:9200/_snapshot/backup/snapshot_1"
```

查看仓库信息

```
curl -XGET 'http://localhost:9200/_snapshot/backup?pretty'
```

rollover

Elasticsearch 从 5.0 开始，为日志场景的用户提供了一个很不错的接口，叫 rollover。其作用是：当某个别名指向的实际索引过大的时候，自动将别名指向下一个实际索引。

因为这个接口是操作的别名，所以我们依然需要首先自己创建一个开始滚动的起始索引：

```
# curl -XPUT 'http://localhost:9200/logstash-2016.11.25-1' -d '{
    "aliases": {
        "logstash": {}
    }
}'
```

然后就可以尝试发起 rollover 请求了：

```
# curl -XPOST 'http://localhost:9200/logstash/_rollover' -d '{
    "conditions": {
        "max_age":   "1d",
        "max_docs":  10000000
    }
}'
```

上面的定义意思就是：当索引超过 1 天，或者索引内的数据量超过一千万条的时候，自动创建并指向下一个索引。

这时候有几种可能性：

- 条件都没满足，直接返回一个 false，索引和别名都不发生实际变化；

```
{
  "old_index" : "logstash-2016.11.25-1",
  "new_index" : "logstash-2016.11.25-1",
  "rolled_over" : false,
  "dry_run" : false,
  "acknowledged" : false,
  "shards_acknowledged" : false,
  "conditions" : {
      "[max_docs: 10000000]" : false,
      "[max_age: 1d]" : false
  }
}
```

- 还没满一天，满了一千万条，那么下一个索引名会是： `logstash-2016.11.25-000002` ；
- 还没满一千万条，满了一天，那么下一个索引名会是： `logstash-2016.11.26-000002` 。

shrink

Elasticsearch 一直以来都是固定分片数的。这个策略极大的简化了分布式系统的复杂度，但是在一些场景，比如存储 metric 的 TSDB、小数据量的日志存储，人们会期望在多分片快速写入数据以后，把老数据合并存储，节约过多的 cluster state 容量。从 5.0 版本开始，Elasticsearch 新提供了 shrink 接口，可以成倍数的合并分片数。

注：所谓成倍数的，就是原来有 *15* 个分片，可以合并缩减成 *5* 个或者 *3* 个或者 *1* 个分片。

整个合并缩减的操作流程，大概如下：

1. 先把所有主分片都转移到一台主机上；
2. 在这台主机上创建一个新索引，分片数较小，其他设置和原索引一致；
3. 把原索引的所有分片，复制（或硬链接）到新索引的目录下；
4. 对新索引进行打开操作恢复分片数据。
5. (可选)重新把新索引的分片均衡到其他节点上。

准备工作

- 因为这个操作流程需要把所有分片都转移到一台主机上，所以作为 shrink 主机，它的磁盘要足够大，至少要能放得下一整个索引。
- 最好是一整块磁盘，因为硬链接是不能跨磁盘的。靠复制太慢了。
- 开始迁移：

  ```
  # curl -XPUT 'http://localhost:9200/metric-2016.11.25/_settings' -d '
  {
    "settings": {
        "index.routing.allocation.require._name": "shrink_node_name",
        "index.blocks.write": true
    }
  }'
  ```

shrink 操作

```
curl -XPOST 'http://localhost:9200/metric-2016.11.25/_shrink/oldmetric-2016.11.25' -d'
{
    "settings": {
        "index.number_of_replicas": 1,
        "index.number_of_shards": 3
    },
    "aliases": {
        "metric-tsdb": {}
    }
}'
```

这个命令执行完会立刻返回，但是 Elasticsearch 会一直等到 shrink 操作完成的时候，才会真的开始做 replica 分片的分配和重均衡，此前分片都处于 initializing 状态。

注意：*Elasticsearch* 有一个硬编码限制，单个分片内的文档总数不得超过 *2147483519* 个。一般来说这个限制在日志场景下是不太会触发的，但是如果做 *TSDB* 用，则需要多加注意！

Ingest 节点

Ingest 节点是 Elasticsearch 5.0 新增的节点类型和功能。其开启方式为：在 `elasticsearch.yml` 中定义：

```
node.ingest: true
```

Ingest 节点的基础原理，是：节点接收到数据之后，根据请求参数中指定的管道流 id，找到对应的已注册管道流，对数据进行处理，然后将处理过后的数据，按照 Elasticsearch 标准的 indexing 流程继续运行。

创建管道流

```
curl -XPUT http://localhost:9200/_ingest/pipeline/my-pipeline-id
 -d '
{
    "description" : "describe pipeline",
    "processors" : [
        {
            "convert" : {
                "field": "foo",
                "type": "integer"
            }
        }
    ]
}'
```

然后发送端带着这个 `my-pipeline-id` 发请求就好了。示例见本书 beats 章节的介绍。

测试管道流

想知道自己的 ingest 配置是否正确，可以通过仿真接口测试验证一下：

```
curl -XPUT http://localhost:9200/_ingest/pipeline/_simulate -d '
{
    "pipeline" : {
        "description" : "describe pipeline",
        "processors" : [
            {
                "set" : {
                    "field": "foo",
                    "value": "bar"
                }
            }
        ]
    },
    "docs" : [
        {
            "_index": "index",
            "_type": "type",
            "_id": "id",
            "_source": {
                "foo" : "bar"
            }
        }
    ]
}'
```

处理器

Ingest 节点的处理器，相当于 Logstash 的 filter 插件。事实上其主要处理器就是直接移植了 Logstash 的 filter 代码成 Java 版本。目前最重要的几个处理器分别是：

convert

```
{
    "convert": {
        "field" : "foo",
        "type": "integer"
    }
}
```

grok

```
    {
        "grok": {
            "field": "message",
            "patterns": ["my %{FAVORITE_DOG:dog} is colored %{RG
B:color}"]
            "pattern_definitions" : {
                "FAVORITE_DOG" : "beagle",
                "RGB" : "RED|GREEN|BLUE"
            }
        }
    }
```

gsub

```
{
    "gsub": {
        "field": "field1",
        "pattern": "\.",
        "replacement": "-"
    }
}
```

date

```
    {
        "date" : {
            "field" : "initial_date",
            "target_field" : "timestamp",
            "formats" : ["dd/MM/yyyy hh:mm:ss"],
            "timezone" : "Europe/Amsterdam"
        }
    }
```

其他处理器插件

除了内置的处理器之外，还有 3 个处理器，官方选择了以插件性质单独发布，它们是 attachement，geoip 和 user-agent 。原因应该是这 3 个处理器需要额外数据模块，而且处理性能一般，担心拖累 ES 集群。

它们可以和其他普通 ES 插件一样安装：

```
sudo bin/elasticsearch-plugin install ingest-geoip
```

使用方式和其他处理器一样：

```
curl -XPUT http://localhost:9200/_ingest/pipeline/my-pipeline-id
-2 -d '
{
    "description" : "Add geoip info",
    "processors" : [
        {
            "geoip" : {
                "field" : "ip",
                "target_field" : "geo",
                "database_file" : "GeoLite2-Country.mmdb.gz"
            }
        }
    ]
}
```

spark streaming 交互

Apache Spark 是一个高性能集群计算框架，其中 Spark Streaming 作为实时批处理组件，因为其简单易上手的特性深受喜爱。在 es-hadoop 2.1.0 版本之后，也新增了对 Spark 的支持，使得结合 ES 和 Spark 成为可能。

目前最新版本的 es-hadoop 是 2.1.0-Beta4。安装如下：

```
wget http://d3kbcqa49mib13.cloudfront.net/spark-1.0.2-bin-cdh4.t
gz
wget http://download.elasticsearch.org/hadoop/elasticsearch-hado
op-2.1.0.Beta4.zip
```

然后通过 `ADD_JARS=../elasticsearch-hadoop-2.1.0.Beta4/dist/elasticsearch-spark_2.10-2.1.0.Beta4.jar` 环境变量，把对应的 jar 包加入 Spark 的 jar 环境中。

下面是一段使用 spark streaming 接收 kafka 消息队列，然后写入 ES 的配置：

```
import org.apache.spark._
import org.apache.spark.streaming.kafka.KafkaUtils
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
import org.apache.spark.sql._
import org.elasticsearch.spark.sql._
import org.apache.spark.storage.StorageLevel
import org.apache.spark.Logging
import org.apache.log4j.{Level, Logger}

object Elastic {
  def main(args: Array[String]) {
    val numThreads = 1
    val zookeeperQuorum = "localhost:2181"
    val groupId = "test"
```

```
    val topic = Array("test").map((_, numThreads)).toMap
    val elasticResource = "apps/blog"

    val sc = new SparkConf()
                 .setMaster("local[*]")
                 .setAppName("Elastic Search Indexer App")

    sc.set("es.index.auto.create", "true")
    val ssc = new StreamingContext(sc, Seconds(10))
    ssc.checkpoint("checkpoint")
    val logs = KafkaUtils.createStream(ssc,
                                       zookeeperQuorum,
                                       groupId,
                                       topic,
                                       StorageLevel.MEMORY_AND_D
ISK_SER)
                         .map(_._2)

    logs.foreachRDD { rdd =>
      val sc = rdd.context
      val sqlContext = new SQLContext(sc)
      val log = sqlContext.jsonRDD(rdd)
      log.saveToEs(elasticResource)
    }

    ssc.start()
    ssc.awaitTermination()

  }
}
```

注意，代码中使用了 spark SQL 提供的 `jsonRDD()` 方法，如果在对应的 kafka topic 里的数据，本身并不是已经处理好了的 JSON 数据的话，这里还需要自己写一写额外的处理函数，利用 `cast class` 来规范数据。

Shield 权限管理

本节作者：*cameluo*

Shield 是 Elastic 公司官方发布的权限管理产品。其主要特性包括：

- 提供集群节点身份验证和集群数据访问身份验证
- 提供基于身份角色的细粒度资源和行为访问控制，细到索引级别的读写控制
- 提供节点间数据传输通道加密保护输出传输安全
- 提供审计功能
- 以插件的形式发布

License管理策略

Shield 是一款商业产品，不过提供 30 天免费试用，试用期间是全功能的。过期后集群会不再正常工作。

Shield架构

用户认证：

Shield 通过定义一套用户集合来认证用户，采用抽象的域方式定义用户集合，支持本地用户(esusers 域)和 LDAP 用户(含 AD)。

- Shield 提供工具 `./bin/shield/esusers` 用于创建和管理本地用户。
- 集成 LDAP 认证支持映射 LDAP 安全组到 Shield 角色，LDAP 安全组与 Shield 角色可以是多对多的关系。

Shield 支持定义多个认证域，采用order字段进行优先级排序。如一个本地域 esusers，order=1，加一个 LDAP 域，order=2。如果用户不再本地用户域中则在 LDAP 域中查找验证。

- `./config/shield/roles.yml` 文件中定义角色和角色的所拥有的权限。
- `./config/shield/group_to_role_mapping.yml` 文件中定义 LDAP 组到角色映射关系。

节点认证与通道加密：

使用SSL/TLS证书进行相互认证和通讯加密。加密是可选配置。如果不使用，shield 节点之间可以进行简单的密码验证（明文传输）。

授权：

Shield 采用 RBAC 授权模型，数据模型包含如下元素：

- 受保护资源(Secured Resource)：控制用户访问的客体，包括 cluster、index/alias 等等。
- 权能(Priviliege)：用户可以对受保护资源执行的一种或多种操作，如 read, write 等。
- 许可(Permissions)：对被保护的资源拥有的一个或多个权能，如 read on the "products" index。
- 角色(Role)：命名的一组许可。
- 用户(Users)：用户实体，可以被赋予 0 个或多种角色，授权他们对被保护的资源执行各种权能。

审计：

增加认证尝试、授权失败等安全相关事件和活动日志。

安装

1. 安装 License 和 Shield 插件

```
bin/plugin -i elasticsearch/license/latest
bin/plugin -i elasticsearch/shield/latest
```

注意：初次运行 Shield 需要重新启动 ES 集群。后续更新 License(license.json 为 License 文件)就可以在线运行：

```
# curl -XPUT -u admin 'http://127.0.0.1:9200/_licenses' -d @lice
nse.json
```

1. 创建本地管理员：

```
./bin/shield/esusers useradd esadmin -r admin
```

配置

这里使用简单的配置先完成基本验证：使用纯本地用户认证或者使用本地认证 + 基本的 ldap 认证。

ES 配置

在elasticsearch.yml中增加如下配置：

```
hostname_verification: false
#shield.ssl.keystore.path:          /app/elasticsearch/node01.jk
s
#shield.ssl.keystore.password:      xxxxxx
shield:
  authc:
    realms:
      default:
        type: esusers
        order: 1
      ldaprealm:
        type: ldap
        order: 2
        url: "ldap://ldap.example.com:389"
        bind_dn: "uid=ldapuser, ou=users, o=services, dc=example
, dc=com"
        bind_password: changeme
        user_search:
          base_dn: "dc=example,dc=com"
          attribute: uid
        group_search:
          base_dn: "dc=example,dc=com"
        files:
          role_mapping: "/app/elasticsearch/shield/group_to_role
_mapping.yml"
        unmapped_groups_as_roles: false
```

角色配置

根据默认配置文件增减角色和访问控制权限。角色配置文件可以在线修改，保存后立即生效：

> ([INFO][shield.authz.store] [Winky Man] updated roles (roles file [/opt/elasticsearch/config/shield/roles.yml] changed))

注意：如果需要集成kibana认证，用户角色也需要有访问'.kibana'索引的访问权限和cluster:monitor/nodes/info的访问权限，具体参照kibana4角色中的定义，否则用户通过kibana认证后仍然无法访问到数据索引

用户组与角色映射配置

根据默认配置文件增减用户、用户组与角色配置中定义角色的映射关系，可以灵活实现各种需求。LDAP组仅支持安全组，不支持动态组。这个配置文件可以在线修改，保存后立即生效：

> ([INFO][shield.authc.ldap.support] [Vishanti] role mappings file [/opt/elasticsearch/config/shield/group_to_role_mapping.yml] changed for realm [ldap/ldaprealm]. updating mappings...)

测试

```
curl -u username http://127.0.0.1:9200/
```

searchguard 在 Elasticsearch 2.x 上的运用

本节作者：*wdh*

本节内容基于以下软件版本：

- elasticsearch 2.1.1-1
- kibana 4.3.1
- logstash 2.1.1-1

searchguard 2.x 更新后跟 shield 配置上很相似，相比之前的版本简洁很多。

searchguard 优点有：

- 节点之间通过 SSL/TLS 传输
- 支持 JDK SSL 和 Open SSL
- 支持热载入，不需要重启服务
- 支持 kibana4 及 logstash 的配置
- 可以控制不同的用户访问不同的权限
- 配置简单

安装

1. 安装search-guard-ssl

```
bin/plugin install com.floragunn/search-guard-ssl/2.1.1.5
```

1. 安装search-guard-2

```
bin/plugin install com.floragunn/search-guard-2/2.1.1.0-alpha2
```

1. 配置 elasticsearch 支持 ssl

elasticsearch.yml增加以下配置：

```
################################################################
```

```
#############################
#                                    SEARCH GUARD SSL
                            #
#                                      Configuration
                            #
################################################################
#############################

#This will likely change with Elasticsearch 2.2, see [PR 14108](
https://github.com/elastic/elasticsearch/pull/14108)
security.manager.enabled: false

################################################################
#############################
# Transport layer SSL
                            #
#
                            #
################################################################
#############################

# Enable or disable node-to-node ssl encryption (default: true)
#searchguard.ssl.transport.enabled: false
# JKS or PKCS12 (default: JKS)
#searchguard.ssl.transport.keystore_type: PKCS12
# Relative path to the keystore file (mandatory, this stores the
 server certificates), must be placed under the config/ dir
searchguard.ssl.transport.keystore_filepath: node0-keystore.jks
# Alias name (default: first alias which could be found)
searchguard.ssl.transport.keystore_alias: my_alias
# Keystore password (default: changeit)
searchguard.ssl.transport.keystore_password: changeit

# JKS or PKCS12 (default: JKS)
#searchguard.ssl.transport.truststore_type: PKCS12
# Relative path to the truststore file (mandatory, this stores t
he client/root certificates), must be placed under the config/ d
ir
searchguard.ssl.transport.truststore_filepath: truststore.jks
# Alias name (default: first alias which could be found)
```

```
searchguard.ssl.transport.truststore_alias: my_alias
# Truststore password (default: changeit)
searchguard.ssl.transport.truststore_password: changeit
# Enforce hostname verification (default: true)
#searchguard.ssl.transport.enforce_hostname_verification: true
# If hostname verification specify if hostname should be resolve
d (default: true)
#searchguard.ssl.transport.resolve_hostname: true
# Use native Open SSL instead of JDK SSL if available (default:
true)
#searchguard.ssl.transport.enable_openssl_if_available: false

################################################################
#############################
# HTTP/REST layer SSL
                            #
#
                            #
################################################################
#############################
# Enable or disable rest layer security - https, (default: false)

#searchguard.ssl.http.enabled: true
# JKS or PKCS12 (default: JKS)
#searchguard.ssl.http.keystore_type: PKCS12
# Relative path to the keystore file (this stores the server cer
tificates), must be placed under the config/ dir
#searchguard.ssl.http.keystore_filepath: keystore_https_node1.jks

# Alias name (default: first alias which could be found)
#searchguard.ssl.http.keystore_alias: my_alias
# Keystore password (default: changeit)
#searchguard.ssl.http.keystore_password: changeit
# Do the clients (typically the browser or the proxy) have to au
thenticate themself to the http server, default is false
#searchguard.ssl.http.enforce_clientauth: false
# JKS or PKCS12 (default: JKS)
#searchguard.ssl.http.truststore_type: PKCS12
# Relative path to the truststore file (this stores the client c
ertificates), must be placed under the config/ dir
```

```
#searchguard.ssl.http.truststore_filepath: truststore_https.jks
# Alias name (default: first alias which could be found)
#searchguard.ssl.http.truststore_alias: my_alias
# Truststore password (default: changeit)
#searchguard.ssl.http.truststore_password: changeit
# Use native Open SSL instead of JDK SSL if available (default:
true)
#searchguard.ssl.http.enable_openssl_if_available: false
```

1. 增加 searchguard 的管理员帐号配置，同样在 elasticsearch.yml 中，增加以下配置：

```
security.manager.enabled: false
searchguard.authcz.admin_dn:
  - "CN=admin,OU=client,O=client,l=tEst,C=De" #DN
```

1. 重启 elasticsearch

将 node 证书和根证书放在 elasticsearch 配置文件目录下，证书可用 openssl 生成，修改了下官方提供的脚本。

注意：证书中的 *client* 的 *DN* 及 *server* 的 *oid*，证书不正确会导致 *es* 服务起不来。（我曾经用 *ejbca* 生成证书不能使用）

配置

searchguard 主要有5个配置文件在 `plugins/search-guard-2/sgconfig` 下：

1. sg_config.yml:

主配置文件不需要做改动

1. sg_internal_users.yml:

user 文件，定义用户。对于 ELK 我们需要一个 kibana 登录用户和一个 logstash 用户：

```
kibana4:
  hash: $2a$12$xZOcnwYPYQ3zIadnlQIJ0eNhX1ngwMkTN.oMwkKxoGvDVPn4/6
XtO
  #password is: kirk
  roles:
    - kibana4
logstash:
  hash: $2a$12$xZOcnwYPYQ3zIadnlQIJ0eNhX1ngwMkTN.oMwkKxoGvDVPn4/6
XtO
```

密码可用plugins/search-guard-2/tools/hash.sh生成

1. sg_roles.yml:

权限配置文件，这里提供 kibana4 和 logstash 的权限，以下是我修改后的内容，可自行修改该部分内容（插件安装自带的配置文件中权限不够，kibana 会登录不了，shield 中同样的权限却是够了）：

```
sg_kibana4:
  cluster:
      - cluster:monitor/nodes/info
      - cluster:monitor/health
  indices:
    '*':
      '*':
        - indices:admin/mappings/fields/get
        - indices:admin/validate/query
        - indices:data/read/search
        - indices:data/read/msearch
        - indices:admin/get
        - indices:data/read/field_stats
    '?kibana':
      '*':
        - indices:admin/exists
        - indices:admin/mapping/put
        - indices:admin/mappings/fields/get
        - indices:admin/refresh
        - indices:admin/validate/query
        - indices:data/read/get
sg_logstash:
  cluster:
    - indices:admin/template/get
    - indices:admin/template/put
  indices:
    'logstash-*':
      '*':
        - WRITE
        - indices:data/write/bulk
        - indices:data/write/delete
        - indices:data/write/update
        - indices:data/read/search
        - indices:data/read/scroll
        - CREATE_INDEX
```

1. sg_roles_mapping.yml:

定义用户的映射关系，添加 kibana 及 logstash 用户对应的映射：

```
sg_logstash:
  users:
    - logstash
sg_kibana4:
  backendroles:
    - kibana
  users:
    - kibana4
```

1. sg_action_groups.yml:

定义权限

加载配置并启用

```
plugins/search-guard-2/tools/sgadmin.sh -cd plugins/search-guard
-2/sgconfig/ -ks plugins/search-guard-2/sgconfig/admin-keystore.
jks -ts plugins/search-guard-2/sgconfig/truststore.jks  -nhnv
```

（如修改了密码，则需要使用plugins/search-guard-2/tools/sgadmin.sh -h查看对应参数）

注意证书路径，将生成的admin证书和根证书放在sgconfig目录下。

最后，可以尝试登录啦！

登录界面会有验证

帐号：kibana4 密码：kirk

更多的权限配置可以自己研究。

请参考

https://github.com/floragunncom/search-guard/tree/2.1.1

监控方案

Elasticsearch 作为一个分布式系统，监控自然是重中之重。Elasticsearch 本身提供了非常完善的，由浅及深的各种性能数据接口。和数据读写检索接口一样，采用 RESTful 风格。我们可以直接使用 curl 来获取数据，编写监控程序，也可以使用一些现成的监控方案。通常这些方案也是通过接口读取数据，解析 JSON，渲染界面。

本章会先介绍一些常用的监控接口，以及其响应数据的含义。然后再介绍几种常用的开源和商业 Elasticsearch 监控产品。

集群健康状态监控

说到 Elasticsearch 集群监控，首先我们肯定是需要一个从总体意义上的概要。不管是多大规模的集群，告诉我正常还是不正常？没错，集群健康状态接口就是用来回答这个问题的，而且这个接口的信息已经出于意料的丰富了。

命令示例

```
# curl -XGET 127.0.0.1:9200/_cluster/health?pretty
{
  "cluster_name" : "es1003",
  "status" : "green",
  "timed_out" : false,
  "number_of_nodes" : 38,
  "number_of_data_nodes" : 27,
  "active_primary_shards" : 1332,
  "active_shards" : 2381,
  "relocating_shards" : 0,
  "initializing_shards" : 0,
  "unassigned_shards" : 0,
  "number_of_pending_tasks" : 0
  "delayed_unassigned_shards" : 0,
  "number_of_in_flight_fetch" : 0,
  "task_max_waiting_in_queue_millis" : 0,
  "active_shards_percent_as_number" : 100.0
}
```

状态信息

输出里最重要的就是 **status** 这行。很多开源的 ES 监控脚本，其实就是拿这行数据做报警判断。status 有三个可能的值：

- green 绿灯，所有分片都正确运行，集群非常健康。
- yellow 黄灯，所有主分片都正确运行，但是有副本分片缺失。这种情况意味着 ES 当前还是正常运行的，但是有一定风险。注意，在 Kibana4 的 server 端启

动逻辑中，即使是黄灯状态，Kibana 4 也会拒绝启动，死循环等待集群状态变成绿灯后才能继续运行。
- red 红灯，有主分片缺失。这部分数据完全不可用。而考虑到 ES 在写入端是简单的取余算法，轮到这个分片上的数据也会持续写入报错。

对 Nagios 熟悉的读者，可以直接将这个红黄绿灯对应上 Nagios 体系中的 Critical，Warning，OK 。

其他数据解释

- number_of_nodes 集群内的总节点数。
- number_of_data_nodes 集群内的总数据节点数。
- active_primary_shards 集群内所有索引的主分片总数。
- active_shards 集群内所有索引的分片总数。
- relocating_shards 正在迁移中的分片数。
- initializing_shards 正在初始化的分片数。
- unassigned_shards 未分配到具体节点上的分片数。
- delayed_unassigned_shards 延时待分配到具体节点上的分片数。

显然，后面 4 项在正常情况下，一般都应该是 0。但是如果真的出来了长期非 0 的情况，怎么才能知道这些长期 unassign 或者 initialize 的分片影响的是哪个索引呢？本书随后还有有更多接口获取相关信息。不过在集群健康这层，本身就可以得到更详细一点的内容了。

level 请求参数

接口请求的时候，可以附加一个 level 参数，指定输出信息以 indices 还是 shards 级别显示。当然，一般来说，indices 级别就够了。

```
# curl -XGET http://127.0.0.1:9200/_cluster/health?level=indices
{
   "cluster_name": "es1003",
   "status": "red",
   "timed_out": false,
   "number_of_nodes": 38,
   "number_of_data_nodes": 27,
   "active_primary_shards": 1332,
```

```
    "active_shards": 2380,
    "relocating_shards": 0,
    "initializing_shards": 0,
    "unassigned_shards": 1
    "delayed_unassigned_shards" : 0,
    "number_of_in_flight_fetch" : 0,
    "task_max_waiting_in_queue_millis" : 0,
    "active_shards_percent_as_number" : 99.0
    "indices": {
       "logstash-2015.05.31": {
          "status": "green",
          "number_of_shards": 81,
          "number_of_replicas": 0,
          "active_primary_shards": 81,
          "active_shards": 81,
          "relocating_shards": 0,
          "initializing_shards": 0,
          "unassigned_shards": 0
       },
       "logstash-2015.05.30": {
          "status": "red",
          "number_of_shards": 81,
          "number_of_replicas": 0,
          "active_primary_shards": 80,
          "active_shards": 80,
          "relocating_shards": 0,
          "initializing_shards": 0,
          "unassigned_shards": 1
       },
       ...
    }
 }
```

这就看到了，是 *logstash-2015.05.30* 索引里，有一个分片一直未能成功分配，导致集群状态异常的。

不过，一般来说，集群健康接口，还是只用来简单监控一下集群状态是否正常。一旦收到异常报警，具体确定 unassign shard 的情况，更推荐使用 kopf 工具在页面查看。

节点状态监控接口

集群状态是从最上层高度来评估你的集群概况，而节点状态则更底层一些，会返回给你集群里每个节点的统计信息。这个接口的信息极为丰富，从硬件到数据到线程，应有尽有。本节会以单节点为例，分段介绍各部分数据的含义。

首先，通过如下命令获取节点状态：

```
# curl -XGET 127.0.0.1:9200/_nodes/stats
```

节点概要

返回数据的第一部分是节点概要，主要就是节点的主机名，网卡地址和监听端口等。这部分内容除了极少数时候(一个主机上运行了多个 ES 节点)一般没有太大用途。

```
{
   "cluster_name": "elasticsearch_zach",
   "nodes": {
      "UNr6ZMf5Qk-YCPA_L18BOQ": {
         "timestamp": 1477886018477,
         "name": "Zach",
         "transport_address": "192.168.1.131:9300",
         "host": "192.168.1.131",
         "ip": "192.168.1.131:9300",
         "roles": [
             "master",
             "data",
             "ingest"
         ],
...
```

索引信息

这部分内容会列出该节点上存储的所有索引数据的状态统计。

1. 首先是概要：

```
    "indices": {
        "docs": {
           "count": 6163666,
           "deleted": 0
        },
        "store": {
           "size_in_bytes": 2301398179,
           "throttle_time_in_millis": 122850
        },
```

docs.count 是节点上存储的数据条目总数；*store.size_in_bytes* 是节点上存储的数据占用磁盘的实际大小。而 *store.throttle_time_in_millis* 则是 ES 进程在做 segment merge 时出现磁盘限速的时长。如果你在 ES 的日志里经常会看到限速声明，那么这里的数值也会偏大。

1. 写入性能：

```
        "indexing": {
           "index_total": 803441,
           "index_time_in_millis": 367654,
           "index_current": 99,
           "delete_total": 0,
           "delete_time_in_millis": 0,
           "delete_current": 0
           "noop_update_total" : 0,
           "is_throttled" : false,
           "throttle_time_in_millis" : 0
        },
```

indexing.index_total 是一个递增累计数，表示节点完成的数据写入总次数。至于后面又删除了多少，额外记录在 *indexing.delete_total* 里；*indexing.is_throttled* 是 2.0 版之后新增的计数，因为 Elasticsearch 从此开始自动管理 throttle，所以有了这个计数。

1. 读取性能：

```
        "get": {
           "total": 6,
           "time_in_millis": 2,
           "exists_total": 5,
           "exists_time_in_millis": 2,
           "missing_total": 1,
           "missing_time_in_millis": 0,
           "current": 0
        },
```

get 这里显示的是直接使用 `_id` 读取数据的状态。

1. 搜索性能：

```
        "search": {
           "open_contexts": 0,
           "query_total": 123,
           "query_time_in_millis": 531,
           "query_current": 0,
           "fetch_total": 3,
           "fetch_time_in_millis": 55,
           "fetch_current": 0
           "scroll_total" : 0,
           "scroll_time_in_millis" : 0,
           "scroll_current" : 0,
           "suggest_total" : 0,
           "suggest_time_in_millis" : 0,
           "suggest_current" : 0
        },
```

search.open_contexts 表示当前正在进行的搜索，而 *search.query_total* 表示节点启动以来完成过的总搜索数，*search.query_time_in_millis* 表示完成上述搜索数花费时间的总和。显然，`query_time_in_millis/query_total` 越大，说明搜索性能越差，可以通过 ES 的 slowlog，获取具体的搜索语句，做出针对性的优化。

search.fetch_total 等指标含义类似。因为 ES 的搜索默认是 query-then-fetch 式的，所以 fetch 一般是少而快的。如果计算出来 `search.fetch_time_in_millis > search.query_time_in_millis`，说明有人采用了较大的 *size* 参数做分页查

询，通过 slowlog 抓到具体的语句，相机优化成 scan 式的搜索。

1. 段合并性能：

```
        "merges": {
           "current": 0,
           "current_docs": 0,
           "current_size_in_bytes": 0,
           "total": 1128,
           "total_time_in_millis": 21338523,
           "total_docs": 7241313,
           "total_size_in_bytes": 5724869463
           "total_stopped_time_in_millis" : 0,
           "total_throttled_time_in_millis" : 0,
           "total_auto_throttle_in_bytes" : 104857600
        },
```

merges 数据分为两部分，current 开头的是当前正在发生的段合并行为统计；total 开头的是历史总计数。一般来说，作为 Elastic Stack 应用，都是以数据写入压力为主的，merges 相关数据会比较突出。

1. 过滤器缓存：

```
        "query_cache": {
           "memory_size_in_bytes": 48,
           "total_count" : 0,
           "hit_count" : 0,
           "miss_count" : 0,
           "cache_size" : 0,
           "cache_count" : 0,
           "evictions": 0
        },
```

query_cache.memory_size_in_bytes 表示过滤器缓存使用的内存，*query_cache.evictions* 表示因内存满被回收的缓存大小，这个数如果较大，说明你的过滤器缓存大小不足，或者过滤器本身不太适合缓存。比如在 Elastic Stack 场景中常用的时间过滤器，如果使用 `@timestamp:["now-1d" TO "now"]` 这种

表达式的话，需要每次计算 now 值，就没法长期缓存。事实上，Kibana 中通过 timepicker 生成的 filtered 请求里，对 `@timestamp` 部分就并不是直接使用 `"now"`，而是在浏览器上计算成毫秒数值，再发送给 ES 的。

请注意，过滤器缓存是建立在 segment 基础上的，在当天新日志的索引中，存在大量的或多或少的 segments。一个已经 5GB 大小的 segment，和一个刚刚 2MB 大小的 segment，发生一次 *query_cache.evictions* 对搜索性能的影响区别是巨大的。但是节点状态中本身这个计数并不能反应这点区别。所以，尽力减少这个数值，但如果搜索本身感觉不慢，那么有几个也无所谓。

1. id 缓存：

```
          "id_cache": {
             "memory_size_in_bytes": 0
          },
```

id_cache 是 parent/child mappings 使用的内存。不过在 Elastic Stack 场景中，一般不会用到这个特性，所以此处数据应该一直是 0。

1. fielddata：

```
          "fielddata": {
             "memory_size_in_bytes": 0,
             "evictions": 0
          },
```

此处显示 fielddata 使用的内存大小。fielddata 用来做聚合，排序等工作。

注意：*fielddata.evictions* 应该永远是 0。一旦发现这个数据大于 0，请立刻检查自己的内存配置，fielddata 限制以及请求语句。

1. segments：

```
        "segments": {
           "count": 1,
           "memory_in_bytes": 2042,
           "terms_memory_in_bytes" : 1510,
           "stored_fields_memory_in_bytes" : 312,
           "term_vectors_memory_in_bytes" : 0,
           "norms_memory_in_bytes" : 128,
           "points_memory_in_bytes" : 0,
           "doc_values_memory_in_bytes" : 92,
           "index_writer_memory_in_bytes" : 0,
           "version_map_memory_in_bytes" : 0,
           "fixed_bit_set_memory_in_bytes" : 0,
           "max_unsafe_auto_id_timestamp" : -1,
           "file_sizes" : {  }
        },
```

segments.count 表示节点上所有索引的 segment 数目的总和。一般来说，一个索引通常会有 50-150 个 segments。再多就对写入性能有较大影响了(可能 merge 速度跟不上新 segment 出现的速度)。所以，请根据节点上的索引数据正确评估节点 segment 的情况。

segments.memory_in_bytes 表示 segments 本身底层数据结构所使用的内存大小。像索引的倒排表，词典，bloom filter(ES1.4以后已经默认关闭) 等，都是要在内存里的。所以过多的 segments 会导致这个数值迅速变大。

操作系统和进程信息

操作系统信息主要包括 CPU，Loadavg，Memory 和 Swap 利用率，文件句柄等。这些内容都是常见的监控项，本书不再赘述。

进程，即 JVM 信息，主要在于 GC 相关数据。

GC

对不了解 JVM 的 GC 的读者，这里先介绍一下 GC(垃圾收集)以及 GC 对 Elasticsearch 的影响。

Java is a garbage-collected language, which means that the programmer does not manually manage memory allocation and deallocation. The programmer simply writes code, and the Java Virtual Machine (JVM) manages the process of allocating memory as needed, and then later cleaning up that memory when no longer needed. Java 是一个自动垃圾收集的编程语言，启动 JVM 虚拟机的时候，会分配到固定大小的内存块，这个块叫做 heap(堆)。JVM 会把 heap 分成两个组：

- Young 新实例化的对象所分配的空间。这个空间一般来说只有 100MB 到 500MB 大小。Young 空间又分为两个 survivor(幸存)空间。当 Young 空间满，就会发生一次 young gc，还存活的对象，就被移入幸存空间里，已失效的对象则被移除。
- Old 老对象存储的空间。这些对象应该是长期存活而且在较长一段时间内不会变化的内容。这个空间会大很多，在 ES 来说，一节点上可能就有 30GB 内存是这个空间。前面提到的 young gc 中，如果某个对象连续多次幸存下来，就会被移进 Old 空间内。而等到 Old 空间满，就会发生一次 old gc，把失效对象移除。

听起来很美好的样子，但是这些都是有代价的！在 GC 发生的时候，JVM 需要暂停程序运行，以便自己追踪对象图收集全部失效对象。在这期间，其他一切都不会继续运行。请求没有响应，ping 没有应答，分片不会分配……

当然，young gc 一般来说执行极快，没太大影响。但是 old 空间那么大，稍慢一点的 gc 就意味着程序几秒乃至十几秒的不可用，这太危险了。

JVM 本身对 gc 算法一直在努力优化，Elasticsearch 也尽量复用内部对象，复用网络缓冲，然后还提供像 Doc Values 这样的特性。但不管怎么说，gc 性能总是我们需要密切关注的数据，因为它是集群稳定性最大的影响因子。

如果你的 ES 集群监控里发现经常有很耗时的 GC，说明集群负载很重，内存不足。严重情况下，这些 GC 导致节点无法正确响应集群之间的 ping ，可能就直接从集群里退出了。然后数据分片也随之在集群中重新迁移，引发更大的网络和磁盘 IO，正常的写入和搜索也会受到影响。

在节点状态数据中，以下部分就是 JVM 相关的数据：

```
"jvm": {
    "timestamp": 1408556438203,
    "uptime_in_millis": 14457,
    "mem": {
       "heap_used_in_bytes": 457252160,
       "heap_used_percent": 44,
       "heap_committed_in_bytes": 1038876672,
       "heap_max_in_bytes": 1038876672,
       "non_heap_used_in_bytes": 38680680,
       "non_heap_committed_in_bytes": 38993920,
    },
```

首先可以看到的就是 heap 的情况。其中这个 `heap_committed_in_bytes` 指的是实际被进程使用的内存，以 JVM 的特性，这个值应该等于 `heap_max_in_bytes`。`heap_used_percent` 则是一个更直观的阈值数据。当这个数据大于 75% 的时候，ES 就要开始 GC。也就是说，如果你的节点这个数据长期在 75% 以上，说明你的节点内存不足，GC 可能会很慢了。更进一步，如果到 85% 或者 95% 了，估计节点一次 GC 能耗时 10s 以上，甚至可能会发生 OOM 了。

继续看下一段：

```
    "pools": {
      "young": {
        "used_in_bytes": 138467752,
        "max_in_bytes": 279183360,
        "peak_used_in_bytes": 279183360,
        "peak_max_in_bytes": 279183360
      },
      "survivor": {
        "used_in_bytes": 34865152,
        "max_in_bytes": 34865152,
        "peak_used_in_bytes": 34865152,
        "peak_max_in_bytes": 34865152
      },
      "old": {
        "used_in_bytes": 283919256,
        "max_in_bytes": 724828160,
        "peak_used_in_bytes": 283919256,
        "peak_max_in_bytes": 724828160
      }
    }
  },
```

这段里面列出了 young, survivor, 和 old GC 区域的情况，不过一般来说用途不大。再看下一段：

```
     "gc": {
       "collectors": {
         "young": {
           "collection_count": 13,
           "collection_time_in_millis": 923
         },
         "old": {
           "collection_count": 0,
           "collection_time_in_millis": 0
         }
       }
     }
```

这里显示的 young 和 old gc 的计数和耗时。young gc 的 count 一般比较大，这是正常情况。old gc 的 count 应该就保持在比较小的状态，包括耗时的 `collection_time_in_millis` 也应该很小。注意这两个计数都是累计的，所以对于一个长期运行的系统，不能拿这个数值直接做报警的判断，应该是取两次节点数据的差值。有了差值之后，再来看耗时的问题，一般来说，一次 young gc 的耗时应该在 1-2 ms，old gc 在 100 ms 的水平。如果这个耗时有量级上的差距，建议打开 slow-GC 日志，具体研究原因。

线程池信息

Elasticsearch 内部是保持着几个线程池的，不同的工作由不同的线程池负责。一般来说，每个池子的工作线程数跟你的 CPU 核数一样。之前有传言中的优化配置是加大这方面的配置项，其实没有什么实际帮助 —— 能干活的 CPU 就那么些个数。所以这段状态数据目的不是用作 ES 配置调优，而是作为性能监控，方便优化你的读写请求。

ES 在 index、bulk、search、get、merge 等各种操作都有专门的线程池，大家的统计数据格式都是类似的：

```
"index": {
   "threads": 1,
   "queue": 0,
   "active": 0,
   "rejected": 0,
   "largest": 1,
   "completed": 1
}
```

这些数据中，最重要的是 rejected 数据。当线程中中所有的工作线程都在忙，即 active == threads，后续的请求就会暂时放到排队的队列里，即 queue > 0。但是每个线程池的 queue 也是有大小限制的，默认是 100。如果后续请求超过这个大小，意味着 ES 真的接受不过来这个请求了，就会把后续请求 reject 掉。

Bulk Rejections

如果你确实注意到了上面数据中的 rejected，很可能就是你在发送 bulk 写入的时候碰到 HTTP 状态码 429 的响应报错了。事实上，集群的承载能力是有上限的。如果你集群每秒就能写入 10000 条数据，以其浪费内存多放几条数据在排队，还不如直接拒绝掉。至少可以让你知道到瓶颈了。

另外有一点可以指出的是，因为 bulk queue 里的数据是维护在内存中，所以节点发生意外死机的时候，是会丢失的。

如果你碰到 bulk rejected，可以尝试以下步骤：

1. 暂停所有的写入进程。
2. 从 bulk 响应中过滤出来 rejected 的那部分。因为 bulk index 中可能大部分已经成功了。
3. 重发一次失败的请求。
4. 恢复写入进程，或者重新来一次上述步骤。

大家可能看出来了，没错，对 rejected 其实压根没什么特殊的操作，重试一次而已。

当然，如果这个 rejected 是持续存在并增长的，那重试也无济于事。你可能需要考虑自己集群是否足以支撑当前的写入速度要求。

如果确实没问题，那么可能是因为客户端并发太多，超过集群的 bulk threads 总数了。尝试减少自己的写入进程个数，改成加大每次 bulk 请求的 size。

文件系统和网络

数据继续往下走，是文件系统和网络的数据。文件系统方面，不管是剩余空间还是 IO 数据，都推荐大家还是通过更传统的系统层监控手段来做。而网络数据方面，主要有两部分内容：

```
        "transport": {
            "server_open": 13,
            "rx_count": 11696,
            "rx_size_in_bytes": 1525774,
            "tx_count": 10282,
            "tx_size_in_bytes": 1440101928
         },
         "http": {
            "current_open": 4,
            "total_opened": 23
         },
```

我们知道 ES 同时运行着 transport 和 http ，默认分别是 9300 和 9200 端口。由于 ES 使用了一些 transport 连接来维护节点内部关系，所以 `transport.server_open` 正常情况下一直会有一定大小。而 `http.current_open` 则是实际连接上来的 HTTP 客户端的数量，考虑到 HTTP 建联的消耗，强烈建议大家使用 keep-alive 长连接的客户端。

Circuit Breaker

继续往下，是 circuit breaker 的数据，包括 request、fielddata、in_flight_requests 和 parent 四种：

```
         "in_flight_requests": {
            "maximum_size_in_bytes": 623326003,
            "maximum_size": "594.4mb",
            "estimated_size_in_bytes": 0,
            "estimated_size": "0b",
            "overhead": 1.03,
            "tripped": 0
         }
```

in_flight_requests 是 5.0 版本新加入的一个控制。在过去版本中，索引速度较慢，而入口流量过大会导致 Client 节点在分发 bulk 流量的时候没有限速而 OOM，现在可以直接对过大的流量返回失败了。

ingest

最后是 ingest 节点独有的 ingest 状态数据。

```
"ingest" : {
    "total" : {
        "count" : 0,
        "time_in_millis" : 0,
        "current" : 0,
        "failed" : 0
    },
    "pipelines" : {
        "set-something" : {
            "count" : 0,
            "time_in_millis" : 0,
            "current" : 0,
            "failed" : 0
        }
    }
}
```

会列出每个定义好的 pipeline 以及最终总体的 ingest 处理量、当前处理中的数据量和处理耗时等。

hot_threads 状态

除了 stats 信息以外，`/_nodes/` 下还有另一个监控接口：

```
# curl -XGET 'http://127.0.0.1:9200/_nodes/_local/hot_threads?interval=60s'
```

该接口会返回在 `interval` 时长内，该节点消耗资源最多的前三个线程的堆栈情况。这对于性能调优初期，采集现状数据，极为有用。

默认的采样间隔是 500ms，一般来说，这个时间范围是不太够的，建议至少 60s 以上。

默认的，资源消耗是按照 CPU 来衡量，还可以用 `?type=wait` 或者 `?type=block` 来查看在等待和堵塞状态的当前线程排名。

索引状态监控接口

索引状态监控接口的输出信息和节点状态监控接口非常类似。一般情况下，这个接口单独监控起来的意义并不大。

不过在 ES 1.6 版开始，加入了对索引分片级别的 commit id 功能。

回忆一下之前原理章节的内容，commit 是在分片内部，对每个 segment 做的。而数据在主分片和副本分片上，是由各自节点自行做 segment merge 操作，所以副本分片和主分片的 segment 的 commit id 是不一致的。这导致 ES 副本恢复时，跟主分片比对 commit id，基本上每个 segment 都不一样，所以才需要从主分片完整重传一份数据。

新加入分片级别的 commit id 后，副本恢复时，先比对跟主分片的分片级 commit id，如果一致，直接本地恢复副本分片内容即可。

查看分片级别 commit id 的命令如下：

```
# curl 'http://127.0.0.1:9200/logstash-mweibo-2015.06.15/_stats/
commit?level=shards&pretty'
...
  "indices" : {
    "logstash-2015.06.15" : {
      "primaries" : { },
      "total" : { },
      "shards" : {
        "0" : [ {
          "routing" : {
            "state" : "STARTED",
            "primary" : true,
            "node" : "AqaYWFQJRIK0ZydvVgASEw",
            "relocating_node" : null
          },
          "commit" : {
            "generation" : 726,
            "user_data" : {
              "translog_id" : "1434297603053",
              "sync_id" : "AU4LEh6wnBE6n0qcEXs5"
            },
            "num_docs" : 36792652
          }
        } ],
...
```

注意：为了节约频繁变更的资源消耗，ES 并不会实时更新分片级 commit id。只有连续 5 分钟没有新数据写入的索引，才会触发给索引各分片更新 commit id 的操作。如果你查看的是一个还在新写入数据的索引，看到的内容应该是下面这样：

```
          "commit" : {
            "generation" : 590,
            "user_data" : {
              "translog_id" : "1434038402801"
            },
            "num_docs" : 29051938
          }
```

任务管理

任务是 Elasticsearch 中早就有的一个概念。不过最新的 5.0 版对此重构之前，我们只能看到对于 master 来说等待执行的集群级别的任务。这个是一个非常狭隘的概念。重构以后，和数据相关的一些操作，也可以以任务形态存在，从而也就有了针对性的管理操作。目前，还只有 recovery、snapshot、reindex 等操作是基于任务式的。未来的 6.0 版，可能把整个 query 检索都改为基于任务式的。困扰用户多年的资源隔离问题，可能就可以得到大大缓解。

等待执行的任务列表

首先我们还是先了解一下狭义的任务，即过去就有的 master 节点的等待执行任务。之前章节已经讲过，master 节点负责处理的任务其实很少，只有集群状态的数据维护。所以绝大多数情况下，这个任务列表应该都是空的。

```
# curl -XGET http://127.0.0.1:9200/_cluster/pending_tasks
{
   "tasks": []
}
```

但是如果你碰上集群有异常，比如频繁有索引映射更新，数据恢复啊，分片分配或者初始化的时候反复出错啊这种时候，就会看到一些任务列表了：

```
{
  "tasks" : [ {
    "insert_order": 767003,
    "priority": "URGENT",
    "source": "create-index [logstash-2015.06.01], cause [api]",
    "time_in_queue_millis": 86,
    "time_in_queue": "86ms"
  }, {
    "insert_order" : 767004,
    "priority" : "HIGH",
    "source" : "shard-failed ([logstash-2015.05.31][50], node[F3
EGSRWGSvWGFlcZ6K9pRA], [P], s[INITIALIZING]), reason [shard fail
```

```
ure [failed recovery][IndexShardGatewayRecoveryException[[logsta
sh-2015.05.31][50] failed to fetch index version after copying i
t over]; nested: CorruptIndexException[[logstash-2015.05.31][50]
 Preexisting corrupted index [corrupted_nC9t_ramRHqsbQqZO78KVg]
caused by: CorruptIndexException[did not read all bytes from fil
e: read 269 vs size 307 (resource: BufferedChecksumIndexInput(NI
OFSIndexInput(path=\"/data1/elasticsearch/data/es1003/nodes/0/in
dices/logstash-2015.05.31/50/index/_16c.si\")))]\norg.apache.luc
ene.index.CorruptIndexException: did not read all bytes from fil
e: read 269 vs size 307 (resource: BufferedChecksumIndexInput(NI
OFSIndexInput(path=\"/data1/elasticsearch/data/es1003/nodes/0/in
dices/logstash-2015.05.31/50/index/_16c.si\")))\n\tat org.apache
.lucene.codecs.CodecUtil.checkFooter(CodecUtil.java:216)\n\tat o
rg.apache.lucene.codecs.lucene46.Lucene46SegmentInfoReader.read(
Lucene46SegmentInfoReader.java:73)\n\tat org.apache.lucene.index
.SegmentInfos.read(SegmentInfos.java:359)\n\tat org.apache.lucen
e.index.SegmentInfos$1.doBody(SegmentInfos.java:462)\n\tat org.a
pache.lucene.index.SegmentInfos$FindSegmentsFile.run(SegmentInfo
s.java:923)\n\tat org.apache.lucene.index.SegmentInfos$FindSegme
ntsFile.run(SegmentInfos.java:769)\n\tat org.apache.lucene.index
.SegmentInfos.read(SegmentInfos.java:458)\n\tat org.elasticsearc
h.common.lucene.Lucene.readSegmentInfos(Lucene.java:89)\n\tat or
g.elasticsearch.index.store.Store.readSegmentsInfo(Store.java:15
8)\n\tat org.elasticsearch.index.store.Store.readLastCommittedSe
gmentsInfo(Store.java:148)\n\tat org.elasticsearch.index.engine.
InternalEngine.flush(InternalEngine.java:675)\n\tat org.elastics
earch.index.engine.InternalEngine.flush(InternalEngine.java:593)
\n\tat org.elasticsearch.index.shard.IndexShard.flush(IndexShard
.java:675)\n\tat org.elasticsearch.index.translog.TranslogServic
e$TranslogBasedFlush$1.run(TranslogService.java:203)\n\tat java.
util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.
java:1142)\n\tat java.util.concurrent.ThreadPoolExecutor$Worker.
run(ThreadPoolExecutor.java:617)\n\tat java.lang.Thread.run(Thre
ad.java:745)\n]; ]]",
    "executing" : true,
    "time_in_queue_millis" : 2813,
    "time_in_queue" : "2.8s"
  }, {
    "insert_order" : 767005,
    "priority" : "HIGH",
```

```
    "source" : "refresh-mapping [logstash-2015.06.03][[curl_debu
ginfo]]",
    "executing" : false,
    "time_in_queue_millis" : 2787,
    "time_in_queue" : "2.7s"
  }, {
    "insert_order" : 767006,
    "priority" : "HIGH",
    "source" : "refresh-mapping [logstash-2015.05.29][[curl_debu
ginfo]]",
    "executing" : false,
    "time_in_queue_millis" : 448,
    "time_in_queue" : "448ms"
  } ]
}
```

可以看到列表中的任务都有各自的优先级，URGENT 优先于 HIGH。然后是任务在队列中的排队时间，任务的具体内容等。

在上例中，由于磁盘文件损坏，一个分片中某个 segment 的实际内容和长度对不上，导致分片数据恢复无法正常完成，堵塞了后续的索引映射更新操作。这个错误一般来说不太常见，也只能是关闭索引，或者放弃这部分数据。更常见的可能，是集群存储长期数据导致索引映射数据确实大到了 master 节点内存不足以快速处理的地步。

这时候，根据实际情况，可以有以下几种选择：

- 索引就是特别多：给 master 加内存。
- 索引里字段太多：改用 nested object 方式节省字段数量。
- 索引多到内存就是不够了：把一部分数据拆出来另一个集群。

新版任务管理

新版本的任务并没有独立的创建接口，你发起的具体某次 snapshot、reindex 等操作，自动就成为了一个任务。而任务的列表可以通过 `/_tasks` 或者 `/_cat/tasks` 接口来获取。和其他接口一样，手工操作选用 cat ，写程序的时候选用 JSON 接口。

```
curl -XGET 'localhost:9200/_cat/tasks?v'
action                         task_id                           parent
_task_id             type      start_time    timestamp running_t
ime ip        node
cluster:monitor/tasks/lists    -ANcpn_JTI-Zs93fGAfhjw:779 -
                     transport 1477891751674 13:29:11  170.2micr
os  127.0.0.1 -ANcpn_
cluster:monitor/tasks/lists[n] -ANcpn_JTI-Zs93fGAfhjw:780 -ANcpn
_JTI-Zs93fGAfhjw:779 direct    1477891751674 13:29:11  60.6micro
s   127.0.0.1 -ANcpn_
indices:data/write/reindex     r1A2WoRbTwKZ516z6NEs5A:916 -
                     transport 1477891751674 13:29:11  212.5micr
os  127.0.0.1 r1A2WoR
```

上面是一个正常运行中的集群的任务列表。除了一个 reindex 任务，没有什么 recovery 的麻烦事儿，很好。

如果想要取消某个任务，比如上面的 reindex，可以像这样运行：

```
curl -XPOST 'localhost:9200/_tasks/task_id:916/_cancel'
```

目前来说，能做的只有这些了。Elasticsearch 还不支持诸如挂起、暂停之类更复杂的任务操作。让我们期待未来的发展吧。

cat 接口的命令行使用

之前介绍的各种接口数据，其响应数据都是 JSON 格式，更适用于程序处理。对于我们日常运维，在 Linux 命令行终端环境来说，简单的分行和分列表格才是更方便的样式。为此，Elasticsearch 提供了 cat 接口。

cat 接口可以读取各种监控数据，可用接口列表如下：

- /_cat/nodes
- /_cat/shards
- /_cat/shards/{index}
- /_cat/aliases
- /_cat/aliases/{alias}
- /_cat/tasks
- /_cat/master
- /_cat/plugins
- /_cat/fielddata
- /_cat/fielddata/{fields}
- /_cat/pending_tasks
- /_cat/count
- /_cat/count/{index}
- /_cat/snapshots/{repository}
- /_cat/recovery
- /_cat/recovery/{index}
- /_cat/segments
- /_cat/segments/{index}
- /_cat/thread_pool
- /_cat/thread_pool/{thread_pools}/_cat/nodeattrs
- /_cat/allocation
- /_cat/repositories
- /_cat/health
- /_cat/indices
- /_cat/indices/{index}

集群状态

还是以最基础的集群状态为例，采用 cat 接口查询集群状态的命令如下：

```
# curl -XGET http://127.0.0.1:9200/_cat/health
1434300299 00:44:59 es1003 red 39 27 2589 1505 4 1 0 0 - 100.0%
```

如果单看这行输出，或许不熟悉的用户会有些茫然。可以通过添加 `?v` 参数，输出表头：

```
# curl -XGET http://127.0.0.1:9200/_cat/health?v
epoch      timestamp cluster status node.total node.data shards
 pri relo init unassign pending_tasks max_task_wait_time active_
shards_percent
1434300334 00:45:34  es1003  green          39         27   2590
1506    5    0        0             0                  -
        100.0%
```

节点状态

```
# curl -XGET http://127.0.0.1:9200/_cat/nodes?v
host      ip           heap.percent ram.percent load_1m load_5m
 load_15m node.role master name
esnode109 10.19.0.109             62          69 6.37
          d         -      10.19.0.109
esnode096 10.19.0.96              63          69 0.29
          -         -      10.19.0.96
esnode100 10.19.0.100             56          79 0.07
          -         m      10.19.0.100
```

跟集群状态不一样的是，节点状态数据太多，cat 接口不方便在一行表格中放下所有数据。所以默认的返回，只是最基本的内存和负载数据。具体想看某方面的数据，也是通过请求参数的方式额外指明。比如想看 heap 百分比和最大值：

```
# curl -XGET 'http://127.0.0.1:9200/_cat/nodes?v&h=ip,port,heapP
ercent,heapMax'
ip            port heapPercent heapMax
192.168.1.131 9300          66 25gb
```

`h` 请求参数可用的值，可以通过 `?help` 请求参数来查询：

```
# curl -XGET http://127.0.0.1:9200/_cat/nodes?help
id                 | id,nodeId                 | unique node id
host               | h                         | host name
ip                 | i                         | ip address
port               | po                        | bound transport por
t
heap.percent       | hp,heapPercent            | used heap ratio
heap.max           | hm,heapMax                | max configured heap
ram.percent        | rp,ramPercent             | used machine memory
 ratio
ram.max            | rm,ramMax                 | total machine memor
y
load               | l                         | most recent load av
g
node.role          | r,role,dc,nodeRole        | d:data node, c:clie
nt node
...
```

中间第二列就是对应的请求参数的值及其缩写。也就是说上面示例还可以写成：

```
# curl -XGET http://127.0.0.1:9200/_cat/nodes?v&h=i,po,hp,hm
```

索引状态

查询索引列表和存储的数据状态是也是 cat 接口最常用的功能之一。为了方便阅读，默认输出时会把数据大小以更可读的方式自动换算成合适的单位，比如 *3.2tb* 这样。

如果你打算通过 shell 管道做后续处理，那么可以加上 `?bytes` 参数，指明统一采用字节数输出，这样保证在同一个级别上排序：

```
# curl -XGET http://127.0.0.1:9200/_cat/indices?bytes=b | sort -rnk8
green open logstash-mweibo-2015.06.12        26 1  754641614   0
 2534955821580 1256680767317
green open logstash-mweibo-2015.06.14        27 1  855516794   0
 2419569433696 1222355996673
```

分片状态

```
# curl -XGET http://127.0.0.1:9200/_cat/shards?v
index                             shard prirep state
docs    store ip        node
logstash-mweibo-h5view-2015.06.10 20    p      STARTED       469
0968  679.2mb 127.0.0.1 10.19.0.108
logstash-mweibo-h5view-2015.06.10 20    r      STARTED       469
0968  679.4mb 127.0.0.1 10.19.0.39
logstash-mweibo-h5view-2015.06.10 2     p      STARTED       472
5961  684.3mb 127.0.0.1 10.19.0.53
logstash-mweibo-h5view-2015.06.10 2     r      STARTED       472
5961  684.3mb 127.0.0.1 10.19.0.102
```

同样，可以用 `?help` 查询其他可用数据细节。比如每个分片的 *segment.count*：

```
# curl -XGET 'http://127.0.0.1:9200/_cat/shards/logstash-mweibo-nginx-2015.06.14?v\&h=n,iic,sc'
n           iic sc
10.19.0.72    0 42
10.19.0.41    0 36
10.19.0.104   0 32
10.19.0.102   0 40
```

恢复状态

在出现集群状态波动时，通过这个接口查看数据迁移和恢复速度也是一个非常有用的功能。不过默认输出是把集群历史上所有发生的 recovery 记录都返回出来，所以一般会加上 `?active_only` 参数，仅列出当前还在运行的恢复状态：

```
# curl -XGET 'http://127.0.0.1:9200/_cat/recovery?active_only&v&
h=i,s,shost,thost,fp,bp,tr,trp,trt'
i                           s  shost     thost     fp    bp    tr
 trp    trt
logstash-mweibo-2015.06.12 10 esnode041 esnode080 87.6% 35.3% 0
 100.0% 0
logstash-mweibo-2015.06.13 10 esnode108 esnode080 95.5% 88.3% 0
 100.0% 0
logstash-mweibo-2015.06.14 17 esnode102 esnode080 96.3% 92.5% 0
 0.0%   118758
```

线程池状态

```
curl -s -XGET http://127.0.0.1:9200/_cat/thread_pool?v
node_name   name                active queue rejected
esnode073   bulk                     1     0    20669
esnode073   fetch_shard_started      0     0        0
esnode073   fetch_shard_store        0     0        0
esnode073   flush                    0     0        0
esnode073   force_merge              0     0        0
esnode073   generic                  0     0        0
esnode073   get                      0     0        0
esnode073   index                    0     0       50
esnode073   listener                 0     0        0
esnode073   management               1     0        0
esnode073   refresh                  0     0        0
esnode073   search                   4     0        0
esnode073   snapshot                 0     0        0
esnode073   warmer                   0     0        0
```

这个接口的输出形式和 5.0 之前的版本有了较大变化，把不同类型的线程状态做了一次行列转换，大大减少了列数以后，对人眼更加合适了。

日志记录

Elasticsearch 作为一个服务，本身也会记录很多日志信息。默认情况下，日志都放在 `$ES_HOME/logs/` 目录里。

日志配置在 Elasticsearch 5.0 中改成了使用 `log4j2.properties` 文件配置，包括日志滚动的方式、命名等，都和标准的 log4j2 一样。唯一的特点是：Elasticsearch 导出了一个变量叫 `${sys:es.logs}`，指向你在 `elasticsearch.yml` 中配置的 `path.logs` 地址：

```
appender.index_search_slowlog_rolling.filePattern = ${sys:es.log
s}_index_search_slowlog-%d{yyyy-MM-dd}.log
```

具体的级别等级也可以通过 `/_cluster/settings` 接口动态调整。比如说，如果你的节点一直无法正确的加入集群，你可以将集群自动发现方面的日志级别修改成 DEBUG，来关注这方面的问题：

```
# curl -XPUT http://127.0.0.1:9200/_cluster/settings -d'
{
    "transient" : {
        "logger.org.elasticsearch.indices.recovery" : "DEBUG"
    }
}'
```

性能日志

除了进程状态的日志输出，ES 还支持跟性能相关的日志输出。针对数据写入，检索，读取三个阶段，都可以设置具体的慢查询阈值，以及不同的输出等级。

此外，慢查询日志是针对索引级别的设置。除了通过 `/_cluster/settings` 接口配置一组集群各索引共用的参数以外，还可以针对每个索引设置不同的参数。

注：过去的版本，还可以在 *`elasticsearch.yml`* 中设置，*5.0* 版禁止在配置文件中添加索引级别的设置！

比如说，我们可以先设置集群共同的参数：

```
# curl -XPUT http://127.0.0.1:9200/_cluster/settings -d'
{
    "transient" : {
        "logger.index.search.slowlog" : "DEBUG",
        "logger.index.indexing.slowlog" : "WARN",
        "index.search.slowlog.threshold.query.debug" : "10s",
        "index.search.slowlog.threshold.fetch.debug": "500ms",
        "index.indexing.slowlog.threshold.index.warn": "5s"
    }
}'
```

然后针对某个比较大的索引，调高设置：

```
# curl -XPUT http://127.0.0.1:9200/logstash-wwwlog-2015.06.21/_s
ettings -d'
{
    "index.search.slowlog.threshold.query.warn" : "10s",
    "index.search.slowlog.threshold.fetch.debug": "500ms",
    "index.indexing.slowlog.threshold.index.info": "10s"
}
```

bigdesk

要想最快的了解 ES 各节点的性能细节，推荐使用 bigdesk 插件，其原作者为 lukas-vlcek。但是从 Elasticsearch 1.4 版本开始就不再更新了。国内有用户 fork 出来继续维护到支持 5.0 版本，GitHub 地址见：https://github.com/hlstudio/bigdesk

bigdesk 通过浏览器直连 ES 节点，发起 RESTful 请求，并渲染结果成图。所以其安装部署极其简单：

```
# git clone https://github.com/hlstudio/bigdesk
# cd bigdesk/_site
# python -mSimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
```

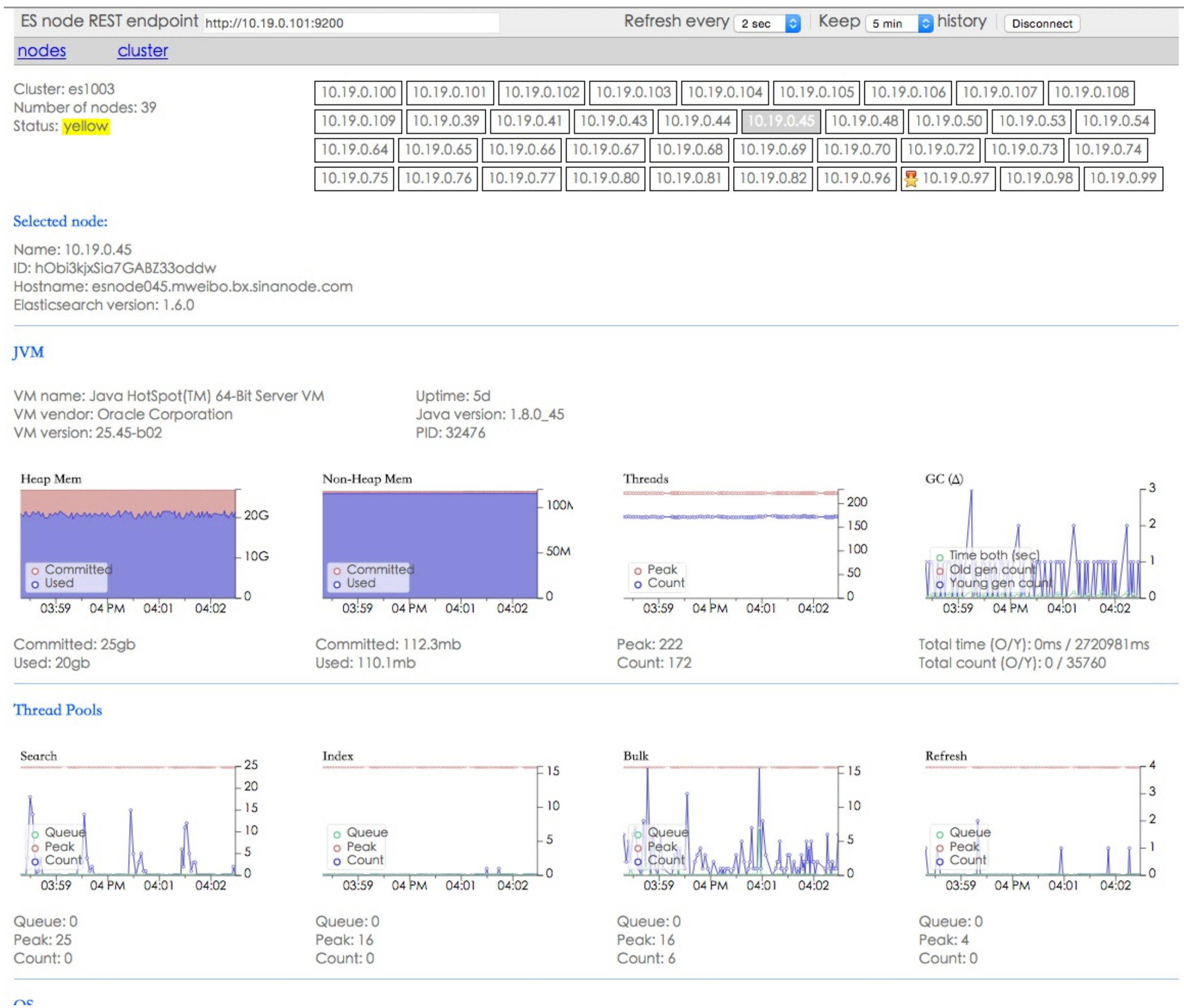
浏览器打开 `http://localhost:8000` 即可看到 bigdesk 页面。在 **endpoint** 输入框内填写要连接的 ES 节点地址，选择 refresh 间隔和 keep 时长，点击 **connect**，完成。



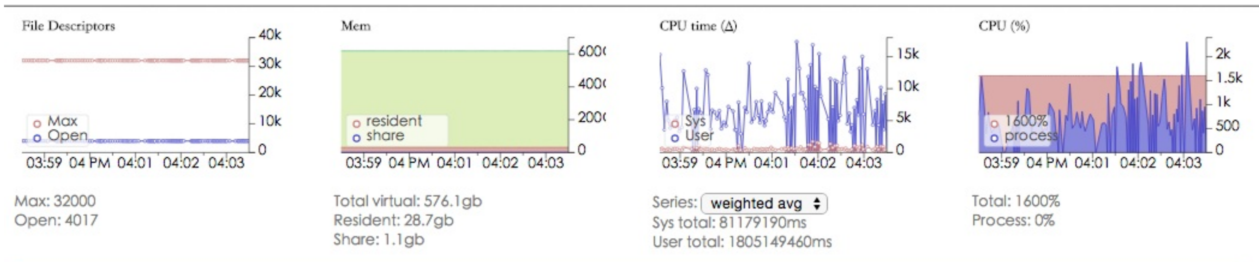
注意：设置 refresh 间隔请考虑 Elastic Stack 使用的 template 里实际的 `refresh_interval` 是多少。否则你可能看到波动太大的数据，不足以说明情况。

点选某个节点后，就可以看到该节点性能的实时走势。一般重点关注 JVM 性能和索引性能。

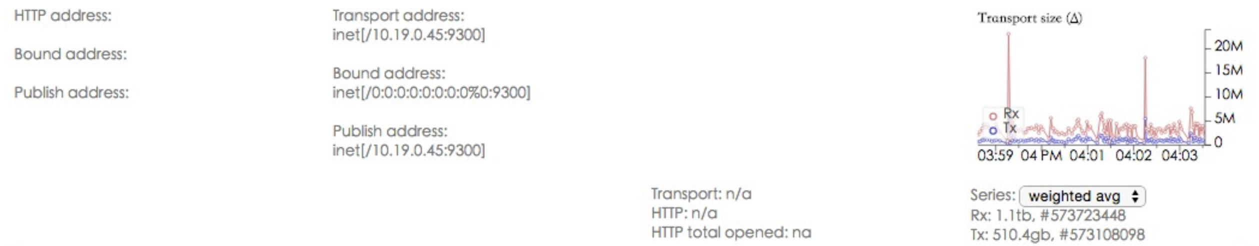
有关 JVM 部分截图如下：



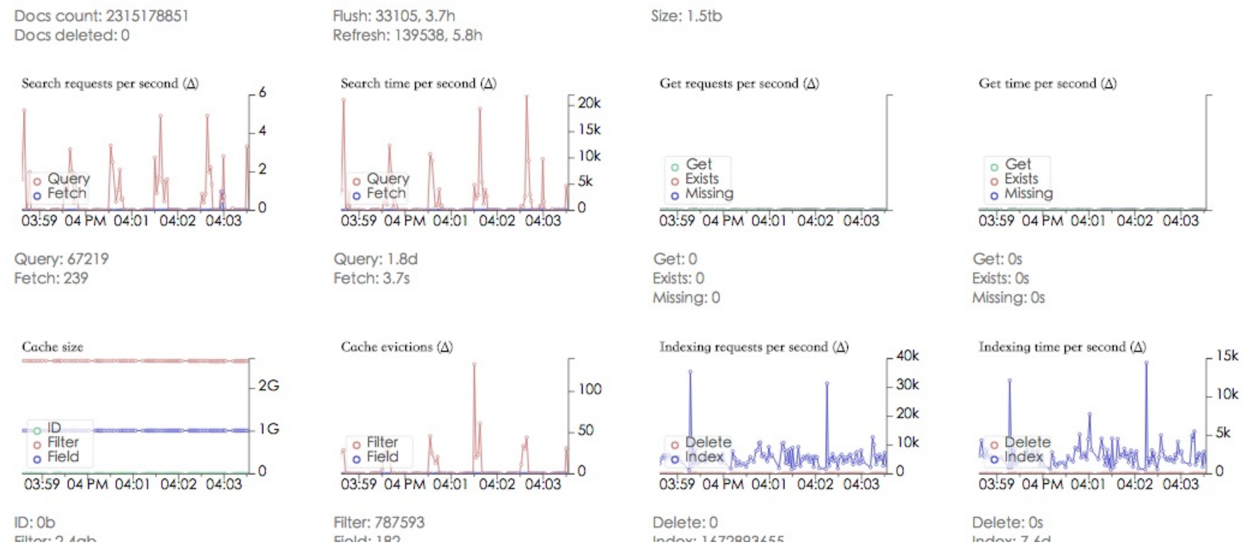
有关数据读写性能部分截图如下：



HTTP & Transport



Indices



cerebro

cerebro 这个名字大家可能觉得很陌生，其实它就是过去的 kopf 插件！因为 Elasticsearch 5.0 不再支持 site plugin，所以 kopf 作者放弃了原项目，另起炉灶搞了 cerebro，以独立的单页应用形式，继续支持新版本下 Elasticsearch 的管理工作。

项目地址：https://github.com/lmenezes/cerebro

安装部署

单页应用的安装方式都非常简单，下载打开即可：

```
# git clone https://github.com/lmenezes/cerebro
# cd cerebro
# ./bin/cerebro
```

然后浏览器打开 `http://localhost:9000` 即可。

zabbix

之前提到的都是 Elasticsearch 的 sites 类型插件，其实质是实时从浏览器读取 cluster stats 接口数据并渲染页面。这种方式直观，但不适合生产环境的自动化监控和报警处理。要达到这个目标，还是需要使用诸如 nagios、zabbix、ganglia、collectd 这类监控系统。

本节以 zabbix 为例，介绍如何使用监控系统完成 Elasticsearch 的监控报警。

github 上有好几个版本的 ESZabbix 仓库，都源自 Elastic 公司员工 untergeek 最早的贡献。但是当时 Elasticsearch 还没有官方 python 客户端，所以监控程序都是用的是 pyes 库。对于最新版的 ES 来说，已经不推荐使用了。

这里推荐一个修改使用了官方 `elasticsearch.py` 库的衍生版。GitHub 地址见：https://github.com/Wprosdocimo/Elasticsearch-zabbix。

安装配置

仓库中包括三个文件：

1. ESzabbix.py
2. ESzabbix.userparm
3. ESzabbix_templates.xml

其中，前两个文件需要分发到每个 ES 节点上。如果节点上运行的是 yum 安装的 zabbix，二者的默认位置应该分别是：

1. `/etc/zabbix/zabbix_externalscripts/ESzabbix.py`
2. `/etc/zabbix/agent_include/ESzabbix.userparm`

然后在各节点安装运行 ESzabbix.py 所需的 python 库依赖：

```
# yum install -y python-pbr python-pip python-urllib3 python-uni
ttest2
# pip install elasticsearch
```

安装成功后，你可以试运行下面这行命令，看看命令输出是否正常：

```
# /etc/zabbix/zabbix_externalscripts/ESzabbix.py cluster status
```

最后一个文件是 zabbix server 上的模板文件，不过在导入模板之前，还需要先创建一个数值映射，因为在模板中，设置了集群状态的触发报警，没有映射的话，报警短信只有 0, 1, 2 数字不是很易懂。

创建数值映射，在浏览器登录 zabbix-web，菜单栏的 **Zabbix Administration** 中选择 **General** 子菜单，然后在右侧下拉框中点击 **Value Maping**。

选择 **create**， 新建表单中填写：

> name: ES Cluster State
> 0 ⇒ Green 1 ⇒ Yellow 2 ⇒ Red

完成以后，即可在 **Templates** 页中通过 **import** 功能完成导入 `ESzabbix_templates.xml` 。

在给 ES 各节点应用新模板之前，需要给每个节点定义一个 `{$NODENAME}` 宏，具体值为该节点 `elasticsearch.yml` 中的 `node.name` 值。从统一配管的角度，建议大家都设置为 ip 地址。

模板应用

导入完成后，zabbix 里多出来三个可用模板：

1. Elasticsearch Node & Cache 其中包括两个 Application：ES Cache 和 ES Node。分别有 Node Field Cache Size, Node Filter Cache Size 和 Node Storage Size, Records indexed per second 共计 4 个 item 监控项。在完成上面说的宏定义后，就可以把这个模板应用到各节点(即监控主机)上了。
2. Elasticsearch Service 只有一个监控项 Elasticsearch service status，做进程监控的，也应用到各节点上。
3. Elasticsearch Cluster 包括 11 个监控项，如下列所示。其中，**ElasticSearch Cluster Status** 这个监控项连带有报警的触发器，并对应之前创建的那个 Value Map。
    - Cluster-wide records indexed per second
    - Cluster-wide storage size
    - ElasticSearch Cluster Status
    - Number of active primary shards

- Number of active shards
- Number of data nodes
- Number of initializing shards
- Number of nodes
- Number of relocating shards
- Number of unassigned shards
- Total number of records 这个模板下都是集群总体情况的监控项，所以，运用在一台有 ES 集群读取权限的主机上即可，比如 zabbix server。

其他

untergeek 最近刚更新了他的仓库，重构了一个 es_stats_zabbix 模块用于 Zabbix 监控，有兴趣的读者可以参考：https://github.com/untergeek/zabbix-grab-bag/blob/master/Elasticsearch/es_stats_zabbix.README.md

Elasticsearch 在运维领域的其他运用

目前 Elasticsearch 虽然以 Elastic Stack 作为主打产品，但其优秀的分布式设计，灵活的搜索评分函数和强大简洁的检索聚合功能，在运维领域也衍生出不少其他有趣的应用方式。

对于 Elastic 公司来说，这些周边应用，也随时可能成为他们的后续目标产品。就在本书第一版编写期间，packetbeat 就被 Elastic 公司收购，并且可能作为未来数据采集端的标准应用。所以，Elastic Stack 用户提前了解其他方面的多种可能，也是非常有意义的。

percolator 接口

在运维体系中，监控和报警总是成双成对的出现。Elastic Stack 在时序统计方面的便捷，在很多时候被作为监控的一种方式在使用。那么，自然就引申出一个问题：Elastic Stack 如何做报警？

对于简单而且固定需求的模式，我们可以在 Logstash 中利用 `filter/metric` 和 `filter/ruby` 等插件做预处理，直接 `output/nagios` 或 `output/zabbix` 来报警；但是对于针对全局的、更复杂的情况，Logstash 就无能为力了。

目前比较通行的办法。有两种：

1. 对于匹配报警，采用 ES 的 Percolator 接口做响应报警；
2. 对于时序统计，采用定时任务方式，发送 ES aggs 请求，分析响应体报警。

针对报警的需求，ES 官方也在最近开发了 Watcher 商业产品，和 Shield 一样以 ES 插件形式存在。本节即稍微描述一下 Percolator 接口的用法和 Watcher 产品的思路。相信稍有编程能力的读者都可以根据自己的需求写出来类似的程序。

Percolator 接口

Percolator 接口和我们习惯的搜索接口正好相反，它要求预先定义好 query，然后通过接口提交文档看能匹配上哪个 query。也就是说，这是一个实时的模式过滤接口。

5.0 版中，对 Percolator 功能做了大幅度改造，现在已经没有单独的接口，而是作为一种 mapping 类型存在。也就是说，我们在创建索引的时候需要预先定义。

比如我们通过 syslog 来发现硬件报错的时候，需要预先定义 mapping：

```
# curl -XPUT http://127.0.0.1:9200/syslog -d '{
    "mappings" : {
        "syslog" : {
            "properties" : {
                "message" : {
                    "type" : "text"
                },
                "severity" : {
                    "type" : "long"
                },
                "program" : {
                    "type" : "keyword"
                }
            }
        },
        "queries" : {
            "properties" : {
                "query" : {
                    "type" : "percolator"
                }
            }
        }
    }
}'
```

然后我们往 `syslog/queries` 里注册 2 条 percolator 请求规则：

```
# curl -XPUT http://127.0.0.1:9200/syslog/queries/memory -d '{
    "query" : {
        "query_string" : {
            "default_field" : "message",
            "default_operator" : "OR",
            "query" : "mem DMA segfault page allocation AND seve
rity:>2 AND program:kernel"
        }
    }
}'
# curl -XPUT http://127.0.0.1:9200/syslog/queries/disk -d '{
    "query" : {
        "query_string" : {
            "default_field" : "message",
            "default_operator" : "OR",
            "query" : "scsi sata hdd sda AND severity:>2 AND pro
gram:kernel"
        }
    }
}'
```

然后，将标准的数据写入请求做一点改动，通过搜索接口进行：

```
# curl -XPOST http://127.0.0.1:9200/syslog/_search -d '{
    "query" : {
        "percolate" : {
            "field" : "query",
            "document_type" : "syslog",
            "document" : {
                "program" : "kernel",
                "severity" : 3,
                "message" : "swapper/0: page allocation failure:
 order:4, mode:0x4020"
            }
        }
    }
}'
```

得到如下结果：

```
{
  ...,
  "hits": [
     {
        "_index": "syslog",
        "_type": "queries",
        "_id": "memory",
        ...
     }
  ]
}
```

从结果可以看出来，这条 syslog 日志匹配上了 memory 异常。接下来就可以发送给报警系统了。

如果是 syslog 索引中已经有的数据，也可以重新过一遍 Percolator 查询。比如我们有一条之前已经写入到 `http://127.0.0.1:9200/syslog/cisco/1234567` 的数据，如下命令就可以把这条数据再过一次 percolate：

```
# curl -XPOST http://127.0.0.1:9200/syslog/_search -d '{
    "query" : {
        "percolate" : {
            "field" : "query",
            "document_type" : "syslog",
            "index" : "syslog",
            "type" : "cisco",
            "id" : "1234567",
        }
    }
}'
```

利用更复杂的 query DSL 做 Percolator 请求的示例，推荐阅读官网这篇 geo 定位的文章：https://www.elastic.co/blog/using-percolator-geo-tagging

Watcher 产品

Watcher 也是 Elastic.co 公司的商业产品，和 Shield，Marvel 一样插件式安装即可：

```
bin/plugin -i elasticsearch/license/latest
bin/plugin -i elasticsearch/watcher/latest
```

Watcher 使用方面，也提供标准的 RESTful 接口，示例如下：

```
# curl -XPUT http://127.0.0.1:9200/_watcher/watch/error_status -
d'
{
    "trigger": {
        "schedule" : { "cron" : "0/5 * * * * ?" }
    },
    "input" : {
        "search" : {
            "request" : {
                "indices" : [ "<logstash-{now/d}>", "<logstash-{
now/d-1d}>" ],
                "body" : {
                    "query" : {
                        "filtered" : {
                            "query" : { "match" : { "status" : "
error" }},
                            "filter" : { "range" : { "@timestamp
" : { "from" : "now-5m" }}}
                        }
                    }
                }
            }
        }
    },
    "condition" : {
        "compare" : { "ctx.payload.hits.total" : { "gt" : 0 }}
    },
```

```
    "transform" : {
        "search" : {
            "request" : {
                "indices" : [ "<logstash-{now/d}>", "<logstash-{
now/d-1d}>" ],
                "body" : {
                    "query" : {
                        "filtered" : {
                            "query" : { "match" : { "status" : "
error" }},
                            "filter" : { "range" : { "@timestamp
" : { "from" : "now-5m" }}}
                        }
                    },
                    "aggs" : {
                        "topn" : {
                            "terms" : {
                                "field" : "userid"
                            }
                        }
                    }
                }
            }
        }
    },
    "actions" : {
        "email_admin" : {
            "throttle_period" : "15m",
            "email" : {
                "to" : "admin@domain",
                "subject" : "Found {{ctx.payload.hits.total}} Er
ror Events at {{ctx.trigger.triggered_time}}",
                "priority" : "high",
                "body" : "Top10 users:\n{{#ctx.payload.aggregati
ons.topn.buckets}}\t{{key}} {{doc_count}}\n{{/ctx.payload.aggreg
ations.topn.buckets}}"
            }
        }
    }
}'
```

上面这行命令，意即:

1. 每 5 分钟，向最近两天的 `logstash-yyyy.MM.dd` 索引发起一次条件为最近五分钟，status 字段内容为 error 的查询请求;
2. 对查询结果做 hits 总数大于 0 的判断;
3. 如果为真，再请求一次上述条件下，userid 字段的 Top 10 数据集作为后续处理的来源;
4. 如果最近 15 分钟内未发送过报警，则向 `admin@domain` 邮箱发送一个标题为 "Found N erroneous events at yyyy-MM-ddTHH:mm:ssZ"，内容为 "Top10 users" 列表的报警邮件。

整个请求体顺序执行。目前 trigger 只支持 scheduler 方式(但是 schedule 下有 crontab、interval、hourly、daily、weekly、monthly、yearly 等多种写法)，input 支持 search 和 http 方式，actions 支持 email，logging，webhook 方式，transform 是可选项，而且可以设置在 actions 里，不同 actions 做不同的 payload 转换。

crontab 定义语法和 Linux 标准不太一致，采用的是 Quartz，文件见：http://www.quartz-scheduler.org/documentation/quartz-1.x/tutorials/crontrigger。

condition, transform 和 actions 中，默认使用 Watcher 增强版的 xmustache 模板语言（示例中的数组循环就是一例）。也可以使用固化的脚本文件，比如有 `threshold_hits.groovy` 的话，可以执行：

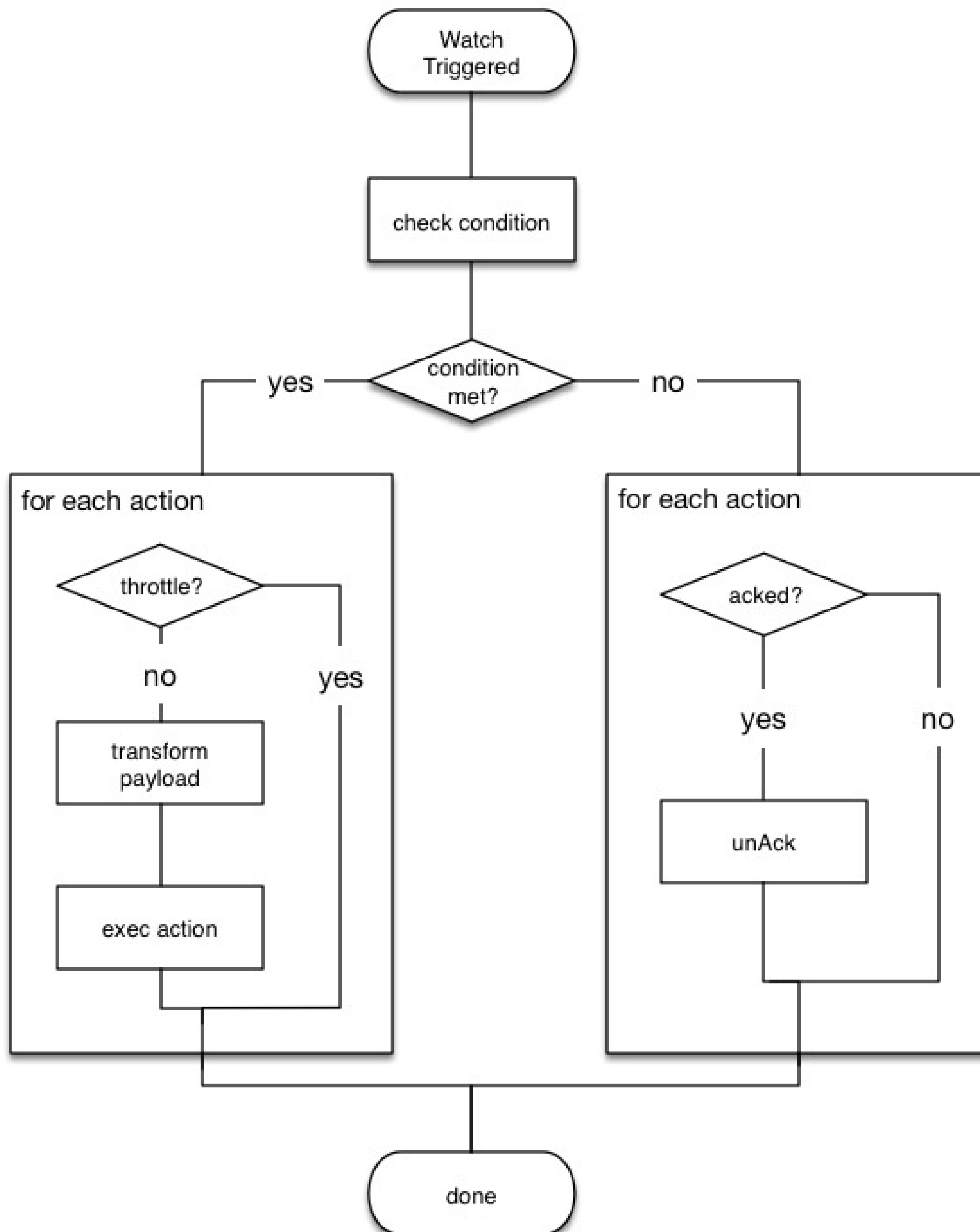
```
"condition" : {
  "script" : {
    "file" : "threshold_hits",
    "params" : {
      "threshold" : 0
    }
  }
}
```

Watcher 中可用的 `ctx` 变量包括：

- `ctx.watch_id`
- `ctx.execution_time`

- `ctx.trigger.triggered_time`
- `ctx.trigger.scheduled_time`
- `ctx.metadata.*`
- `ctx.payload.*`

完整的 Watcher 插件内部执行流程如下图。相信有编程能力的读者都可以用 crontab/at 配合 curl，email 工具仿造出来类似功能的 shell 脚本。



注意：

在 search 中，对 indices 内容可以写完整的索引名比如 `syslog` ，也可以写通配符比如 `logstash-*` ，也可以写时序索引动态定义方式如 `<logstash-{now/d}>` 。而这个动态定义，Watcher 是支持根据时区来确定的，这个需要在 `elasticsearch.yml` 里配置一行：

```
watcher.input.search.dynamic_indices.time_zone: '+08:00'
```

笔者仿照 watcher 的配置语法，开源了一个基于 Kibana 扩展的类 watcher 监控项目，本书稍后 Kibana 章节将会有详细介绍。

ElastAlert

ElastAlert 是 Yelp 公司开源的一套用 Python2.6 写的报警框架。属于后来 Elastic.co 公司出品的 Watcher 同类产品。官网地址见：http://elastalert.readthedocs.org/。

安装

比官网文档说的步骤稍微复杂一点，因为其中 mock 模块安装时依赖的 setuptools 要求版本在 0.17 以上，CentOS6 默认的不够，需要通过 yum 命令升级，当前可以升级到的是 0.18 版。

```
# yum install python-setuptools
# git clone https://github.com/Yelp/elastalert.git
# cd elastalert
# python setup.py install
# cp config.yaml.example config.yaml
```

安装完成后会自带三个命令：

- elastalert-create-index ElastAlert 会把执行记录存放到一个 ES 索引中，该命令就是用来创建这个索引的，默认情况下，索引名叫 `elastalert_status` 。其中有 4 个 _type，都有自己的 `@timestamp` 字段，所以同样也可以用 kibana 来查看这个索引的日志记录情况。
- elastalert-rule-from-kibana 从 Kibana3 已保存的仪表盘中读取 Filtering 设置，帮助生成 `config.yaml` 里的配置。不过注意，它只会读取 filtering，不包括 queries。
- elastalert-test-rule 测试自定义配置中的 rule 设置。

最后，运行命令:

```
# python -m elastalert.elastalert --config ./config.yaml
```

或者单独执行 `rules_folder` 里的某个 rule：

```
# python -m elastalert.elastalert --config ./config.yaml --rule
./examele_rules/one_rule.yaml
```

配置结构

和 Watcher 类似(或者说也只有这种方式)，ElastAlert 配置结构也分几个部分，但是它有自己的命名。

query 部分

除了有关 ES 服务器的配置以外，主要包括：

- `run_every` 配置，用来设置定时向 ES 发请求，默认 5 分钟。
- `buffer_time` 配置，用来设置请求里时间字段的范围，默认 45 分钟。
- `rules_folder` 配置，用来加载下一阶段的 rule 设置，默认是 `example_rules` 。
- `timestamp_field` 配置，设置 `buffer_time` 时针对哪个字段，默认是 `@timestamp` 。
- `timestamp_type` 配置，设置 `timestamp_field` 的时间类型，ElastAlert 内部也需要转换成时间对象，默认是 `ISO8601` ，也可以是 `UNIX` 。

rule 部分

rule 设置各自独立以文件方式存储在 `rules_folder` 设置的目录里。其中可以定义下面这些参数：

- `name` 配置，每个 rule 需要有自己独立的 name，一旦重复，进程将无法启动。
- `type` 配置，选择某一种数据验证方式。
- `index` 配置，从某类索引里读取数据，目前已经支持Ymd格式，需要先设置 use_strftime_index: true，然后匹配索引，配置形如：index: logstash-es-test-%Y.%m.%d，表示匹配logstash-es-test名称开头，以年月日作为索引后缀的 index。
- `filter` 配置，设置向 ES 请求的过滤条件。
- `timeframe` 配置，累积触发报警的时长。

- `alert` 配置，设置触发报警时执行哪些报警手段。

不同的 type 还有自己独特的配置选项。目前 ElastAlert 有以下几种自带 ruletype：

- `any`：只要有匹配就报警；
- `blacklist`：`compare_key` 字段的内容匹配上 `blacklist` 数组里任意内容；
- `whitelist`：`compare_key` 字段的内容一个都没能匹配上 `whitelist` 数组里内容；
- `change`：在相同 `query_key` 条件下，`compare_key` 字段的内容，在 `timeframe` 范围内发送变化；
- `frequency`：在相同 `query_key` 条件下，`timeframe` 范围内有 `num_events` 个被过滤出来的异常；
- `spike`：在相同 `query_key` 条件下，前后两个 `timeframe` 范围内数据量相差比例超过 `spike_height`。其中可以通过 `spike_type` 设置具体涨跌方向是 `up`，`down`，`both`。还可以通过 `threshold_ref` 设置要求上一个周期数据量的下限，`threshold_cur` 设置要求当前周期数据量的下限，如果数据量不到下限，也不触发；
- `flatline`：`timeframe` 范围内，数据量小于 `threshold` 阈值；
- `new_term`：`fields` 字段新出现之前 `terms_window_size` (默认 30 天) 范围内最多的 `terms_size` (默认 50) 个结果以外的数据；
- `cardinality`：在相同 `query_key` 条件下，`timeframe` 范围内 `cardinality_field` 的值超过 `max_cardinality` 或者低于 `min_cardinality`。

alert 部分

`alert` 配置是一个数组，目前支持 command, email，jira，opsgenie，sns，hipchat，slack 等方式。

- command

command 最灵活也最简单。默认会采用 `%(fieldname)s` 格式：

```
command: ["/bin/send_alert", "--username", "%(username)s", "--time", "%(key_as_string)s"]
```

如果要用的比较多，可以开启 `pipe_match_json` 参数，会把整个过滤到的内容，以一整个 JSON 字符串的方式管道输入指定脚本。

- email

email 方式采用 SMTP 协议，所以有一系列 `smtp_*` 配置，然后加上 `email` 参数提供收件人地址数组。

特殊的是，email 和 jira 两种方式，ElastAlert 提供了一些内容格式化模板：

比如可以这样控制邮件标题：

```
alert_subject: "Issue {0} occurred at {1}"
alert_subject_args:
  - issue.name
  - "@timestamp"
```

而默认的邮件内容模板是：

```
body = rule_name
       [alert_text]
       ruletype_text
       {top_counts}
       {field_values}
```

这些内容同样可以通过 `alert_text` (及对应 `alert_text_args`)等来灵活修改。

此外，alert 还有一系列控制报警风暴的选项，从属于 rule：

- `aggregation` ：设置一个时长，则该时长内所有报警最终合在一起发一次；
- `realert` ：设置一个时长，则该时长内，相同 `query_key` 的报警只发一个；
- `exponential_realert` ：设置一个时长，必须大于 `realert` 设置。则在 `realert` 到 `exponential_realert` 之间，每次报警后，realert 自动翻倍。

微信告警插件

社区有人提供了使用微信做 ElastAlert 告警操作的扩展，其 GitHub 地址见：https://github.com/anjia0532/elastalert-wechat-plugin 。

enhancements 部分

`match_enhancements` 配置，设置一个数组，在报警内容发送到 alert 之前修改具体数据。ElastAlert 默认不提供具体的 enhancements 实现，需要自己扩展。

不过，作为通用方式，ElastAlert 提供几个便捷选项，把 Kibana 地址加入报警：

- `generate_kibana_link`：自动生成一个 Kibana3 的临时仪表盘附在报警内容上。
- `use_kibana_dashboard`：采用现成的 Kibana3 仪表盘附在报警内容上。
- `use_kibana4_dashboard`：采用现成的 Kibana4 仪表盘附在报警内容上。

扩展

rule

创建一个自己的 rule，是以 Python 模块的形式存在的，所以首先创建目录：

```
# mkdir rule_modules
# cd rule_modules
# touch __init__.py example_rule.py
```

`example_rule.py` 的内容如下：

```
import dateutil.parser
from elastalert.util import ts_to_dt
from elastalert.ruletypes import RuleType

class AwesomeNewRule(RuleType):
    # 用来指定本 rule 对应的配置文件中必要的参数项
    required_options = set(['time_start', 'time_end', 'usernames
'])
    # 每次运行获取的数据以时间排序数据传递给 add_data 函数
    def add_data(self, data):
        for document in data:
            # 配置文件中的设置可以通过 self.rule[] 获取
            if document['username'] in self.rule['usernames']:
                login_time = document['@timestamp'].time()
                time_start = dateutil.parser.parse(self.rule['ti
me_start']).time()
                time_end = dateutil.parser.parse(self.rule['time
_end']).time()
                if login_time > time_start and login_time < time
_end:
                    # 最终过滤结果，使用 self.add_match 添加
                    self.add_match(document)

    # alert_text 中使用的文本
    def get_match_str(self, match):
        return "%s logged in between %s and %s" % (match['userna
me'],
                                                         self.rule['ti
me_start'],
                                                         self.rule['ti
me_end'])
    def garbage_collect(self, timestamp):
        pass
```

配置中，指定

```
type: rule_modules.example_rule.AwesomeRule
time_start: "20:00"
time_end: "24:00"
usernames:
  - "admin"
  - "userXYZ"
  - "foobaz"
```

即可使用。

alerter

alerter 也是以 Python 模块的形式存在的，所以还是要创建目录(如果之前二次开发 rule 已经创建过可以跳过)：

```
# mkdir rule_modules
# cd rule_modules
# touch __init__.py example_alert.py
```

`example_alert.py` 的内容如下：

```
from elastalert.alerts import Alerter, basic_match_string

class AwesomeNewAlerter(Alerter):
    required_options = set(['output_file_path'])
    def alert(self, matches):
        for match in matches:
            with open(self.rule['output_file_path'], "a") as out
put_file:
                # basic_match_string 函数用来转换异常数据成默认格式的
字符串
                match_string = basic_match_string(self.rule, mat
ch)
                output_file.write(match_string)
    # 报警发出后，ElastAlert 会调用该函数的结果写入 ES 索引的 alert_inf
o 字段内
    def get_info(self):
        return {'type': 'Awesome Alerter',
                'output_file': self.rule['output_file_path']}
```

配置中，指定

```
alert: "rule_modules.example_alert.AwesomeNewAlerter"
output_file_path: "/tmp/alerts.log"
```

即可使用。

enhancement

enhancement 也是以 Python 模块的形式存在的，所以还是要创建目录(如果之前二次开发 rule 或 alert 已经创建过可以跳过)：

```
# mkdir rule_modules
# cd rule_modules
# touch __init__.py example_enhancement.py
```

`example_enhancement.py` 的内容如下：

```
from elastalert.enhancements import BaseEnhancement
class MyEnhancement(BaseEnhancement):
    def process(self, match):
        if 'domain' in match:
            url = "http://who.is/whois/%s" % (match['domain'])
            match['domain_whois_link'] = url
```

在需要的 rule 配置文件中添加如下内容即可启用：

```
match_enhancements:
  - "rule_modules.example_enhancement.MyEnhancement"
```

因为 `match_enhancements` 是个数组，也就是说，如果数组有多个 enhancement，会依次执行，完全完成后，才传递给 alert。

时序数据

之前已经介绍过，ES 默认存储数据时，是有索引数据、*_all* 全文索引数据、*_source* JSON 字符串三份的。其中，索引数据由于倒排索引的结构，压缩比非常高。因此，在某些特定环境和需求下，可以只保留索引数据，以极小的容量代价，换取 ES 灵活的数据结构和聚合统计功能。

在监控系统中，对监控项和监控数据的设计一般是这样：

> metric_path value timestamp (Graphite 设计) { "host": "Host name 1", "key": "item_key", "value": "33", "clock": 1381482894 } (Zabbix 设计)

这些设计有个共同点，数据是二维平面的。以最简单的访问请求状态监控为例，一次请求，可能转换出来的 `metric_path` 或者说 `key` 就有：**{city,isp,host,upstream}.{urlpath...}.{status,rt,ut,size,speed}** 这么多种。假设 urlpath 有 1000 个，就是 20000 个组合。意味着需要发送 20000 条数据，做 20000 次存储。

而在 ES 里，这就是实实在在 1000 条日志。而且在多条日志的时候，因为词元的相对固定，压缩比还会更高。所以，使用 ES 来做时序监控数据的存储和查询，是完全可行的办法。

对时序数据，关键就是定义缩减数据重复。template 示例如下：

```
{
  "order" : 2,
  "template" : "logstash-monit-*",
  "settings" : {
  },
  "mappings" : {
    "_default_" : {
      "_source" : {
        "enabled" : false
      },
      "_all" : {
        "enabled" : false
      }
    }
  },
  "aliases" : { }
}
```

如果有些字段，是完全不用 Query ，只参加 Aggregation 的，还可以设置：

```
      "properties" : {
        "sid" : {
          "index" : "no",
          "type" : "keyword"
        }
      },
```

关于 Elasticsearch 用作 rrd 用途，与 MongoDB 等其他工具的性能测试与对比，可以阅读腾讯工程师写的系列文章：http://segmentfault.com/a/1190000002690600

Grafana

Grafana是一个开源的指标量监测和可视化工具。常用于展示基础设施的时序数据和应用程序运行分析。Grafana的dashboard展示非常炫酷，绝对是运维提升逼格的一大利器。

官方在线的demo可以在这里找到: http://play.grafana.org/

grafana的套路基本上跟kibana差不多，都是根据查询条件设置聚合规则，在合适的图表上进行展示，多个图表共同组建成一个dashboard，熟悉kibana的用户应该可以非常容易上手。另外grafana的可视化功能比kibana强得多，后面逐步会介绍到，而且4以上版本将集成报警功能。

安装

Grafana的安装非常简单，官方就有软件仓库可以直接使用，也可以通过docker镜像等方式直接本地启动。

参考官方文档的安装方法，对应找到你的操作系统的安装方法即可: http://docs.grafana.org/

> 值得一提的是，由于官方仓库托管在S3上，国内用户直接访问苦不堪言，万幸的是清华大学tuna镜像站已经提供了grafana的镜像，只需要将官方文档中提到的仓库地址对应的换成清华大学的镜像站的地址即可: https://mirrors.tuna.tsinghua.edu.cn/grafana/

安装完毕后，使用 `service grafana-server start` 就可以启动grafana，访问 `http://your-host:3000` 就可以看到登录界面了。

默认的用户名和密码都是 `admin` 。

> 默认情况下，grafana的配置存储于 `sqlite3` 中，如果你想使用其他存储后端，如 `mysql` ， `postgresql` 等，请参考官方文档配置: http://docs.grafana.org/installation/configuration/

grafana的几个基本构成和基本概念参考官方文档: http://docs.grafana.org/guides/basic_concepts/ ，后面会逐步提到这些关键词。

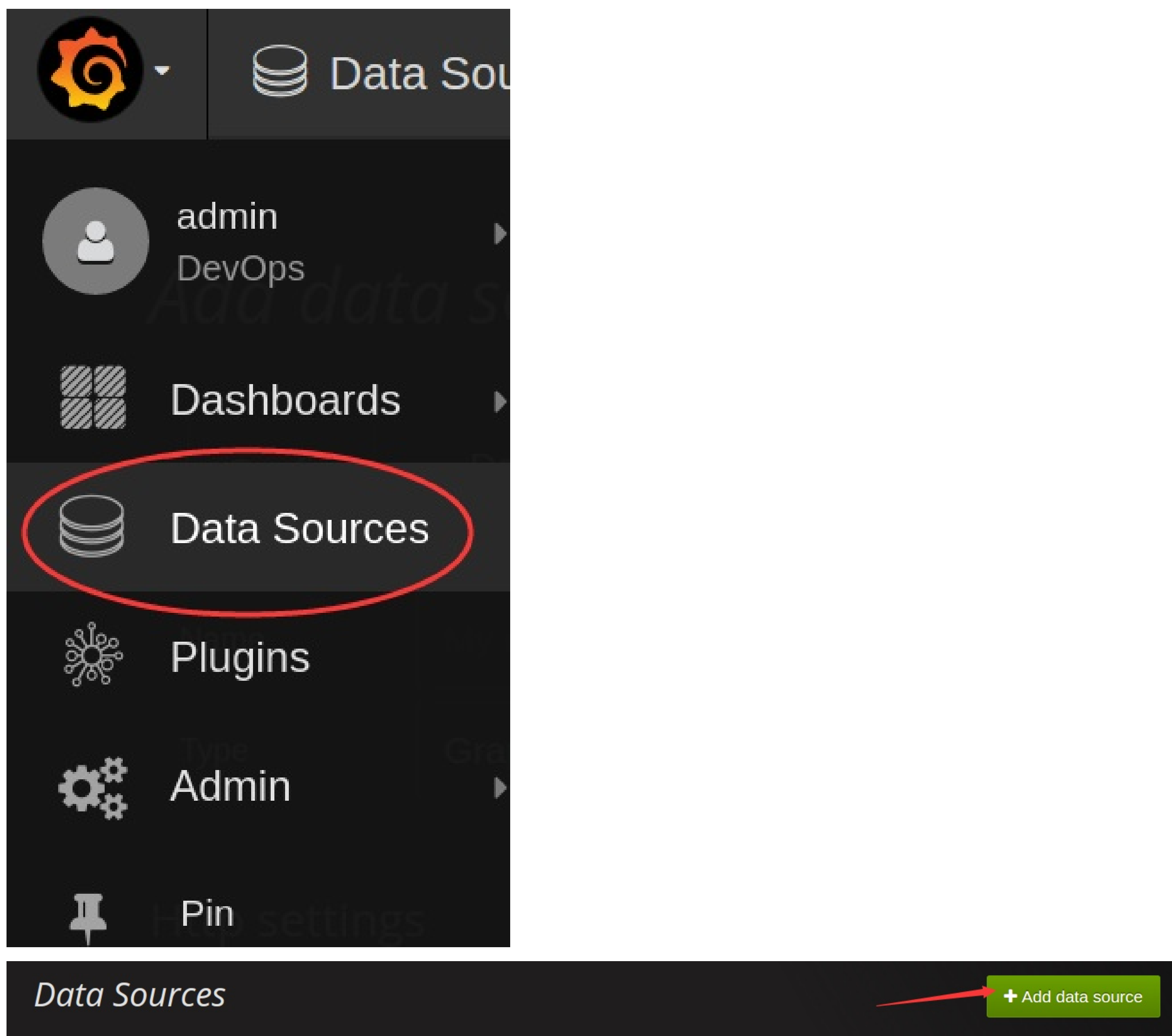
配置数据源

先要明确一点，grafana只是一个dashboard(4版本开始将引入报警功能)，负责把数据库中的数据进行可视化展示，本身并不存储任何数据，另外某些查询和聚合使用的是数据库本身提供的功能，需要对应的数据库去支持。因此不代表你换了一个数据库后端就一定能展示相同的图形。

grafana目前支持的时序数据库有: Graphite, Prometheus, Elasticsearch, InfluxDB, OpenTSDB, AWS Cloudwatch。未来可能会有更多的数据库的支持加入，请关注更新。也可以使用第三方插件引入支持。

我们这里使用 `Elasticsearch` 作为数据库的来源。

首先进入数据源的设置页面，点击左上角的图标，选择 `Data Sources` - `Add data sources` :



进入数据源的设置，例如我们要从 `logstash-YYYY.MM.DD` 这些index中读取数据，就像kibana默认的index那样，像这样配置即可:

Edit data source

Name logstash Default

Type Elasticsearch 数据源选ElasticSearch

Http settings

Url http://localhost:9200

Access proxy 这里注意，如果你的ES不对外暴露，选proxy，上面的Url选择方向代理的后端。否则选direct，Url填入你能访问到的ES地址

Http Auth Basic Auth With Credentials

Elasticsearch details

Index name [logstash-]YYYY.MM.DD Pattern Daily

Time field name @timestamp 这里照着Kibana的配置抄就可以了如果你要用别的index，这里就照着改就行了

Version 2.x 版本要选对，3.1.1版本只支持到2.x，未来版本将会支持5.0

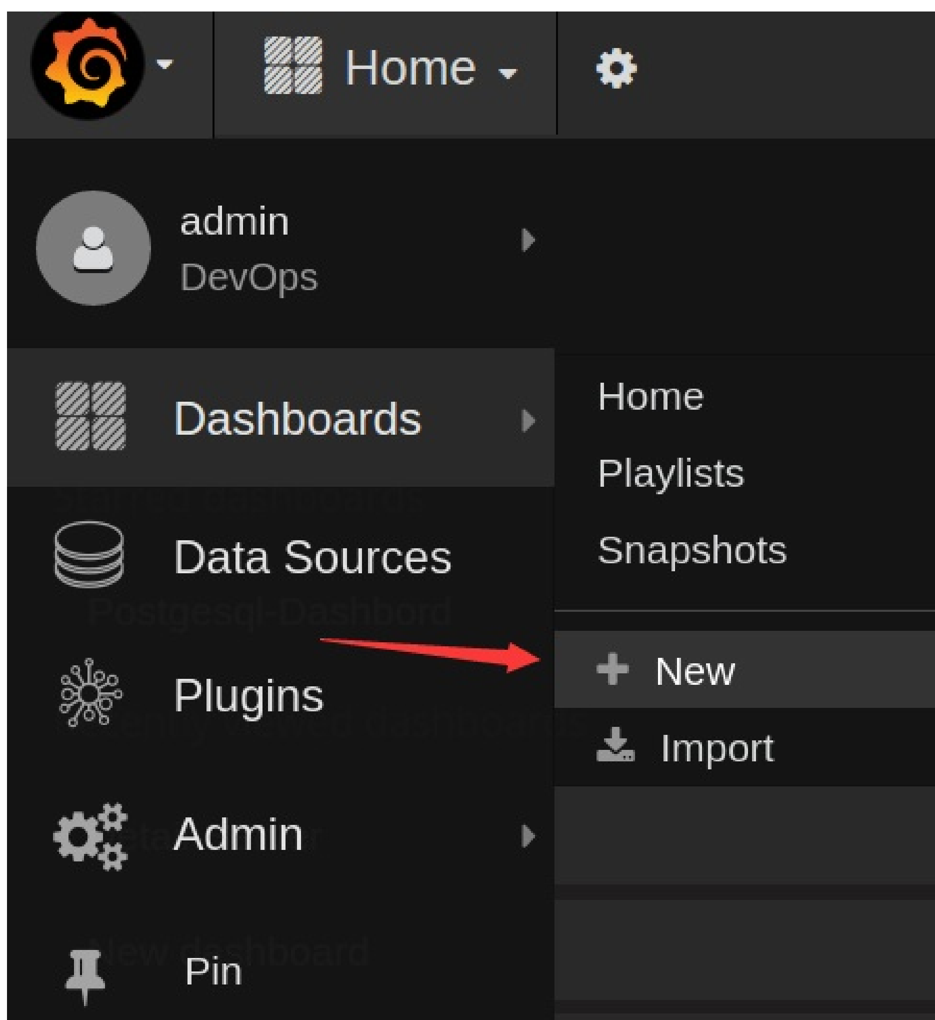
Default query settings

Group by time interval example: >10s

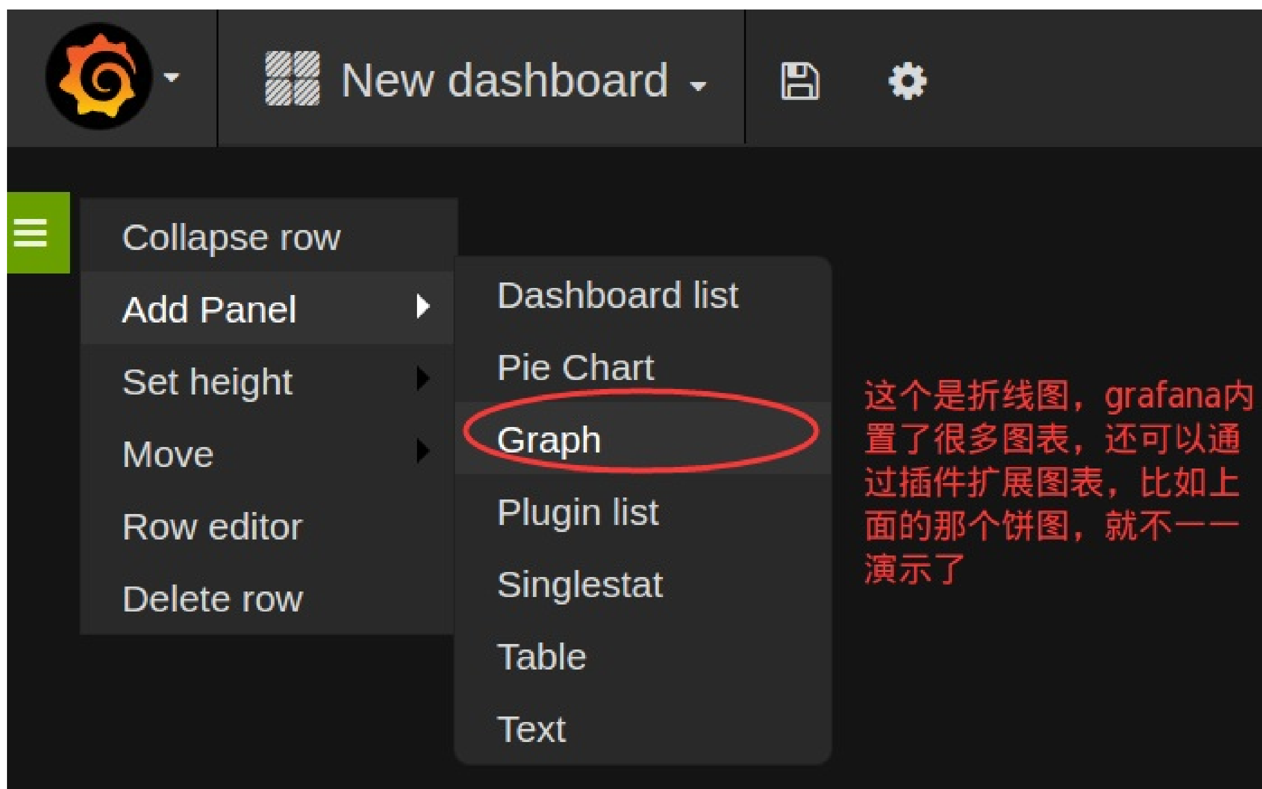
生成第一个图表

数据源配置好之后，就可以开始我们可视化的第一步了。这一步相对较为简单，只要熟悉ES的query-string-syntax(https://www.elastic.co/guide/en/elasticsearch/reference/5.0/query-dsl-query-string-query.html#query-string-syntax)就可以轻松写出来。

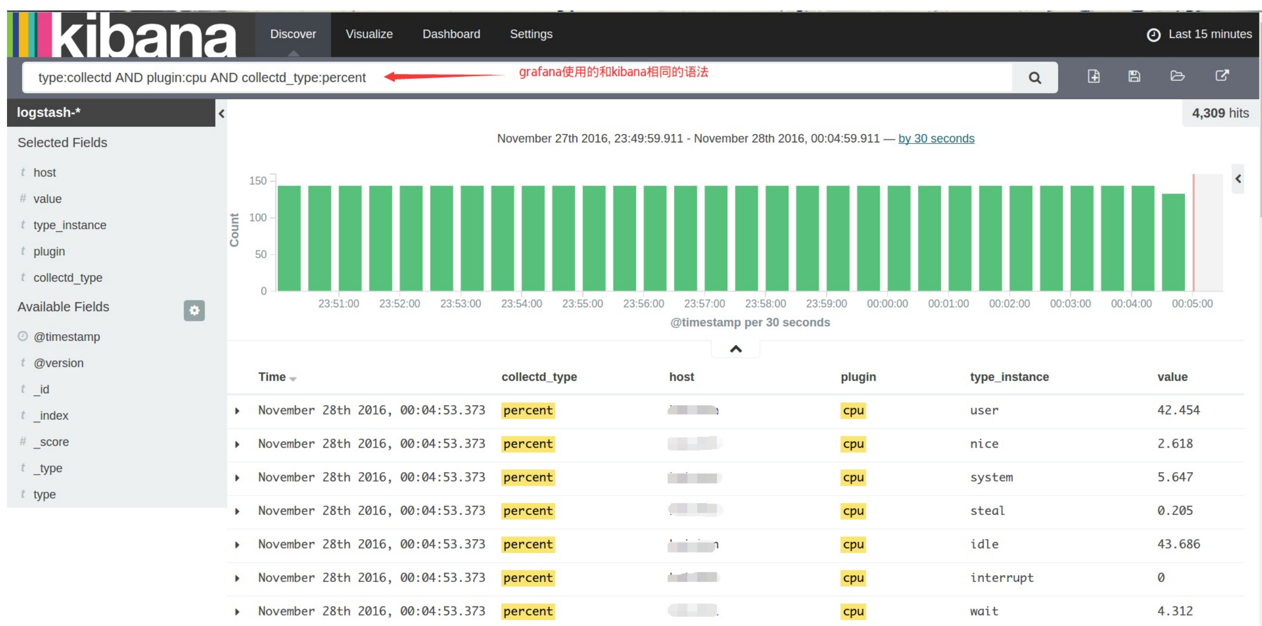
想创建图表，首先得有 `dashboard` ，new一个dashboard出来，就可以尽情的开搞了：



比如我现在通过 `collectd` 收集上来了cpu占用比的数据，希望绘制成CPU曲线，很明显我们需要创建一个 `折线图` ，那么在dashboard上new一个折线图的panel即可(graph):

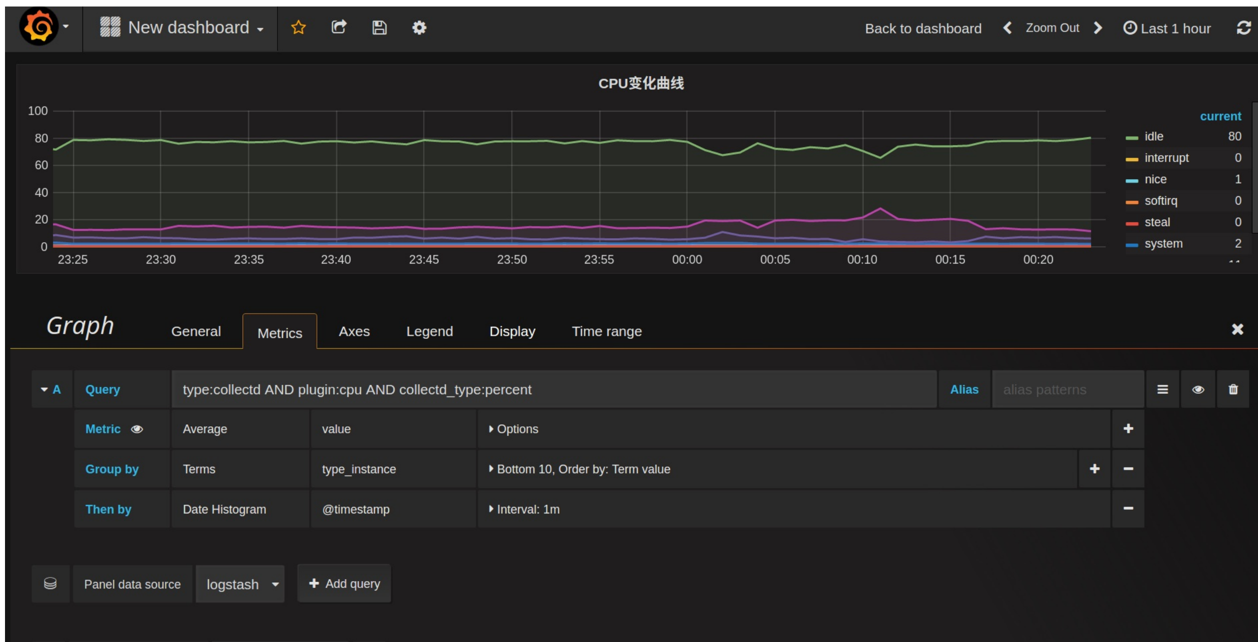


先在kibana中看看查询结果:



很明显， `value` 是希望绘制到曲线上的点，而 `type_instance` 应该作为ES聚合的 `bucket` ，如果 `host` 有多个，那么 `host` 也应作为一个bucket，我们先不管这个host，后面会再说怎么按照host分图形，grafana有更巧妙的办法。

bucket的作用非常类似于sql中的 `GROUP BY` ，于是新建的panel的查询应该像这样:



简单解释一下这些配置:

- **query**: 对应的就是kibana上面的查询条件，先把想要展示的数据全部select出来，然后进行聚合或筛选, **Alias**是给这个查询起个别名，可用于图例说明
- **metric**: 这个就是关键的指标量的，到底展示的是什么东西。这里展示的是GROUP BY之后对value求平均值
- **Group by**: 对应elasticsearch的bucket，按照 `type_instance` 分桶

整个查询下来大致相当于SQL中的 `SELECT avg(value) WHERE query_string('type:collectd AND plugin:cpu AND collectd_type:percent') GROUP BY type_instance GROUP BY Date_Histogram(@timestamp)`

如果只关心CPU的消耗，不关心CPU的空闲(idle)，希望仅仅把 `idle` 曲线隐藏掉，也很简单，只要在 `Display` 的选项卡中选择隐藏掉 `idle` 的line即可，瞧:



graph图表能配置的项很多，包括图例的颜色，位置，显示的值等等，甚至可以配置数据的单位，以便于更优雅的展示，这里就不一一列举了。

把希望展示的图表一个个加进去，最终一个完整的dashboard就生成了：



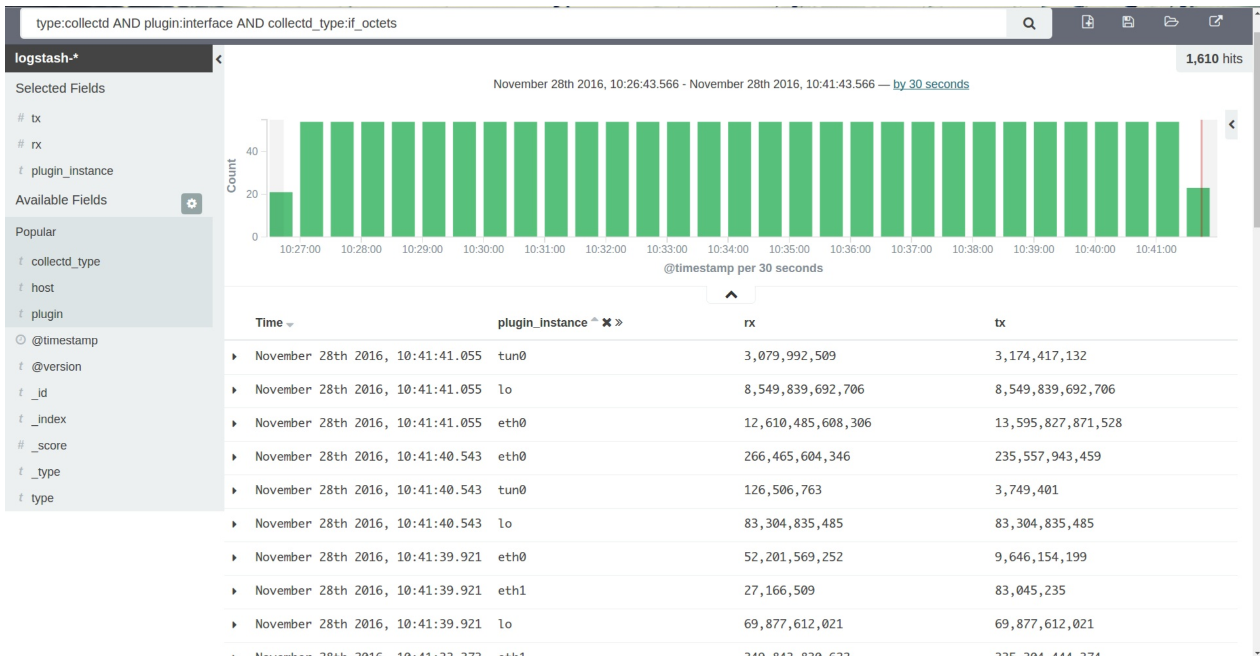
模板功能

这个功能实在是grafana的一大亮点，不得不提。模板可以让你轻松的批量生成同一类型的查询，而不用一个个添加这些panel，实在是生成动态可视化图表的大杀器。

比如现在要统计各个网卡的流量，而服务器可能包含多个网卡，`lo`，`eth0`，`eth1`等等，按照前面的套路，可能会想到将网卡名作为bucket进行GROUP BY操作。思路没错，但是这会导致所有网卡的流量在同一个图表显示，当网卡多的时候非常凌乱。再比如网卡之间可能流量根本不对等，某块网卡的流量高出其他网卡好几个数量级，这将导致其他网卡的流量曲线非常接近0，以至于几乎看不清楚流量变化了。

因此需要将不同的网卡流量放在不同的panel分别展示，但是不同服务器的网卡数量可能又不一样，因此查询条件没法写成一个固定的，需要动态化。

先从kibana看看网卡流量的查询:



查询条件为 `type:collectd AND plugin:interface AND collectd_type:if_octets` ，可以得到所有网卡每一时刻的 `rx` , `tx` 的值。如果我们要得到每一块网卡的流量，很明显要再追加一个动态查询条件 `AND plugin_instance:$interface` ，需要提前对 `plugin_instance` 做一次 `distinct` ，拿出所有可能的值，再拼接到这个动态查询上。这个就是grafana模板的基本使用方式。

要想使用模板，首先要先创建动态参数的查询条件，在顶部点击设置按钮，进入 `Templating` 设置:

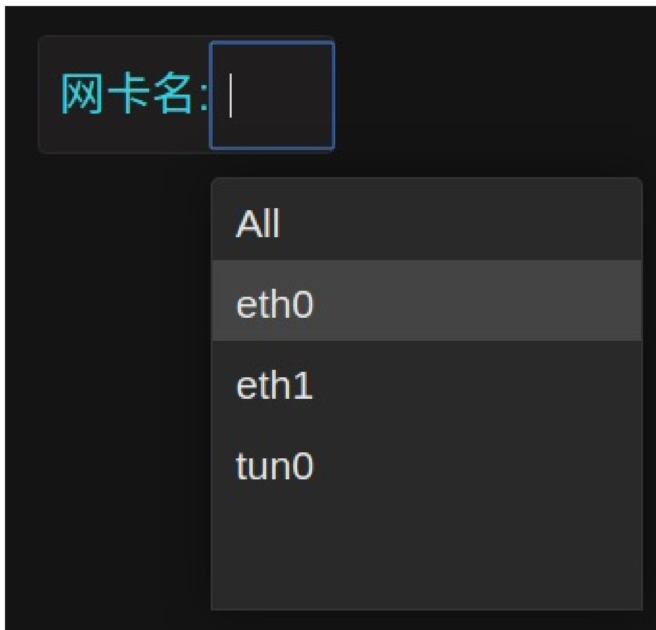


最关键的地方在 `Query` 这个配置，这里就是定义怎么获取到 `$interface` 这个变量的所有可选值，每个数据库的语法不一样，比如Elasticsearch的Template语法在这里: http://docs.grafana.org/datasources/elasticsearch/#templating ，如果你使用

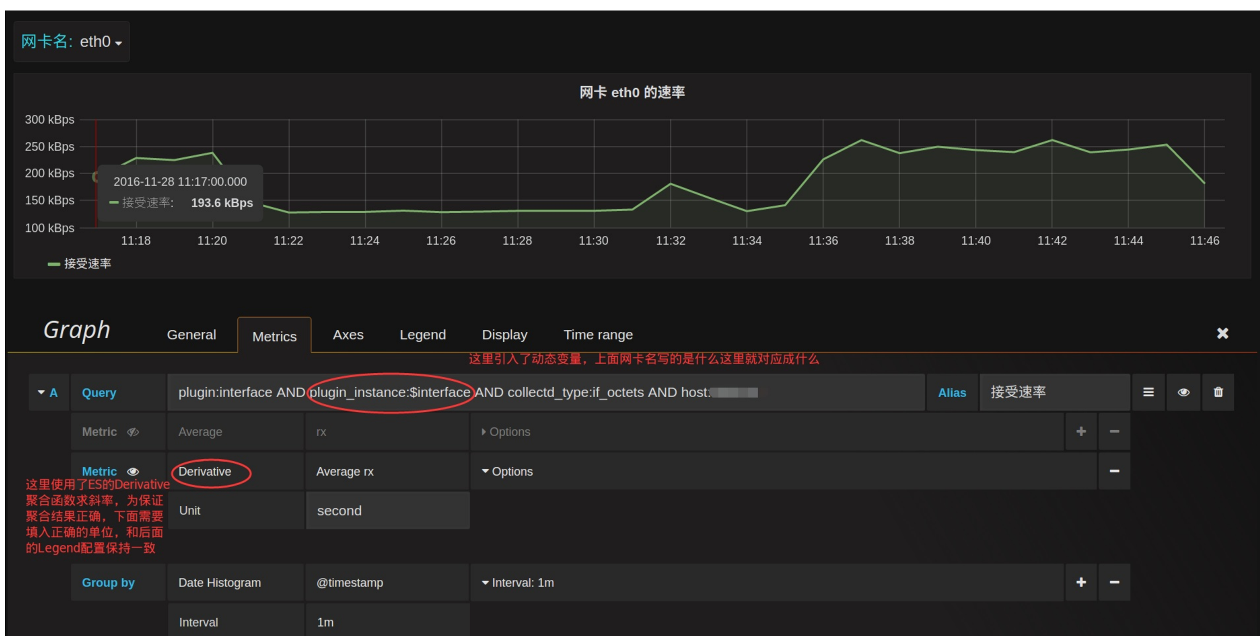
的是其他数据源，那么查询对应数据源的模板语法即可。

这个查询就相当于在 `plugin:interface` 这个查询条件下，对 `plugin_instance` 这个field进行distinct操作，结果再使用 `/[^lo]/` 正则过滤，将 `lo` 这个结果从结果集中排除掉。

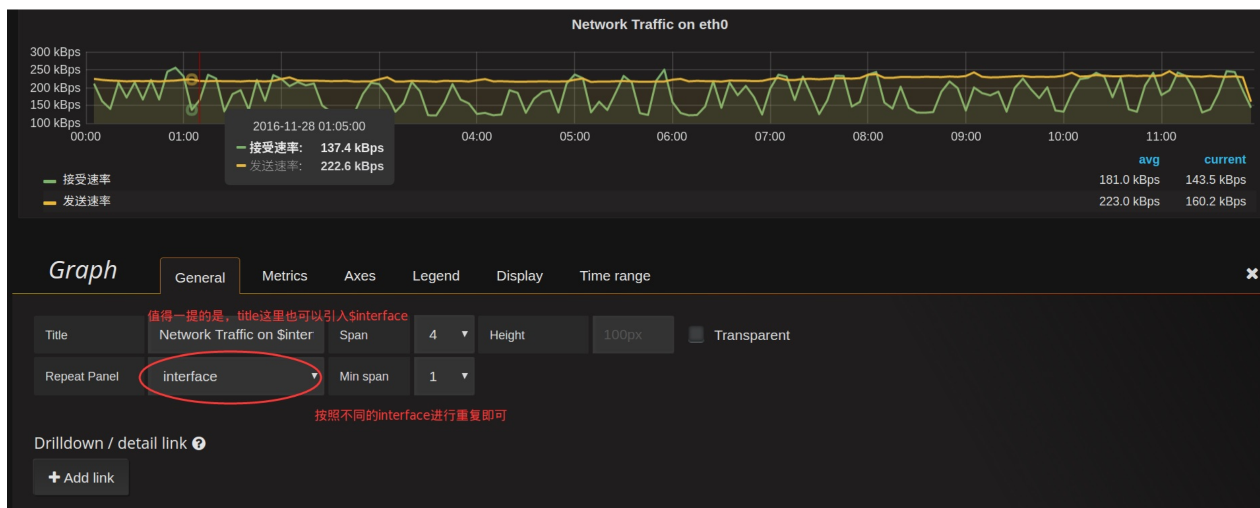
结束后我们就可以看到了这样的单选下拉框:



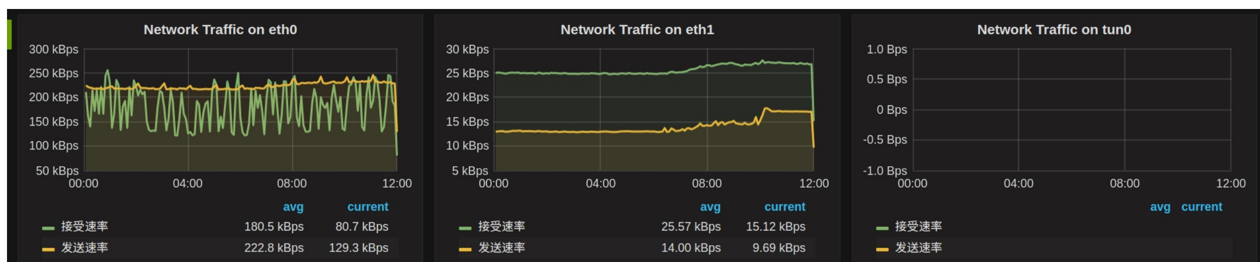
引入了 `$interface` 这个变量后，我们就可以对查询条件做下改进了:



现在这个图形就会跟着上面网卡名的改变而改变了。如果希望选择 `All` 的时候，能同时展示多块网卡，需要把这一个panel进行重复就可以了，在 `General` 选项卡启用 `repeat` :



当网卡名选择 `All` 的时候，就会将所有的网卡流量曲线图repeat成一行(row):



同理，如果 `Templating` 那里将 `Multi-value` 勾选上，那么网卡名这里就可以勾选上你希望展示的网卡名，并不限于单选。

现在再回到一开始遗留的问题，host匹配的问题，利用模板的特性就可以非常优雅的搞定这个问题。

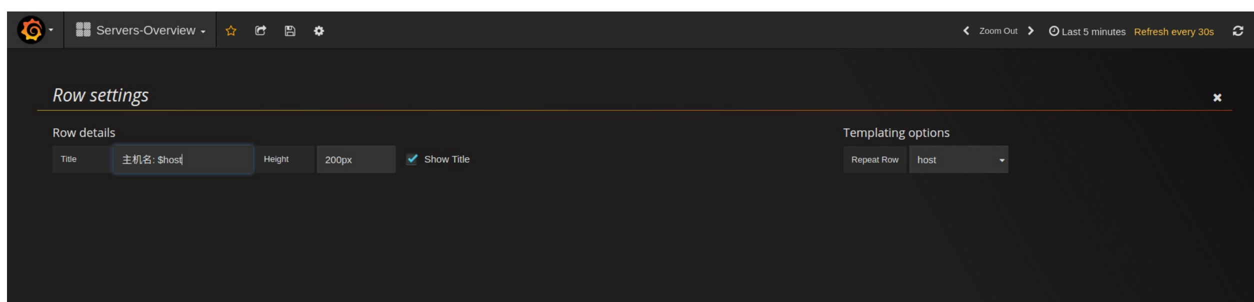
创建一个host的模板:

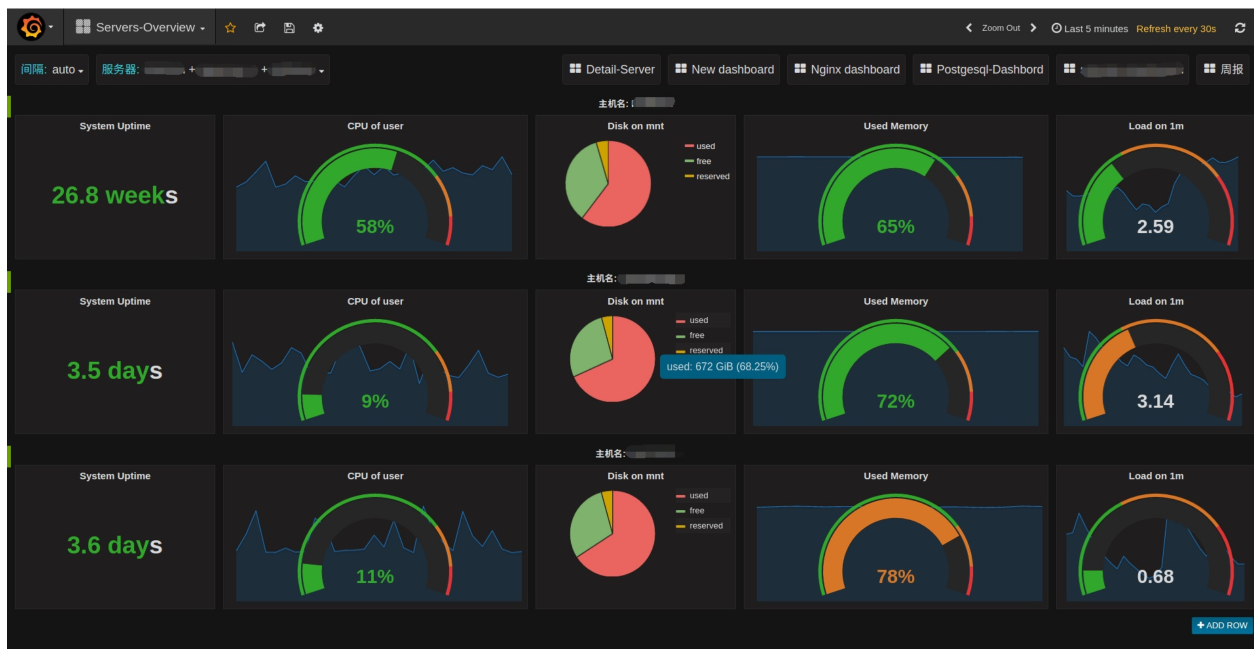
Templating
Variables
Edit
Variable
Name
host
Type
Query
Label
主机名
Hide
Query Options
Data source
default
Refresh
On Dashboard Load
Query
{"find": "terms", "field": "host"}
Regex
/.*-(.*)-.*/
Selection Options
Multi-value
Include All option
Value groups/tags (Experimental feature)
Enable

这样前面所有的查询 `Query` 再追加一个条件 `AND host:$host` ，就可以对每个主机生成一个单独的dashboard界面了：



不止panel可以repeat(似乎只能repeat成一行)，连row都可以repeat:





每行一个服务器的概览图，repeat之后就可以将多个服务器的主要指标放在一个dashborad中了。

grafana的玩法还有很多，期待慢慢发掘，更多详情可以参考官方文档。

在线资源

grafana提供了一些在线资源，可以帮助使用者更方便的使用grafana，比如在线dashboard(https://grafana.net/dashboards)可以帮助快速生成一个美观的dashboard，不用自己花心思去布局了，再比如在线的插件仓库(https://grafana.net/plugins)可以帮助连接其他数据源，如zabbix，Open-Falcon等，或添加其他展示图表，如饼图(Pie chat)。

合理利用这些在线资源可以让grafana更加完善易用。

参考资料

- grafana官方文档: http://docs.grafana.org/
- elasticsearch官方文档: https://www.elastic.co/guide/en/elasticsearch/reference/2.4/index.html

juttle 介绍

juttle 是一个 nodejs 项目，专注于数据处理和可视化。它自定义了一套自己的 DSL，提供交互式命令行、程序运行、界面访问三种运行方式。

在 juttle 的 DSL 中，可以用 `|` 管道符串联下列指令实现数据处理：

- 通过 read 指令读取来自 http、file、elasticsearch、graphite、influxdb、opentsdb、mysql 等数据源，
- 通过 filter 指令及自定义的 JavaScript 函数做数据过滤，
- 通过 reduce 指令做数据聚合，
- 通过 join 指令做数据关联，
- 通过 write 指令做数据转储，
- 通过 view 指令做数据可视化。

更关键的，可以用 `()` 并联同一层级的多条指令进行处理。

看起来非常有意思的项目，赶紧试试吧。

安装部署

既然说了这是一个 nodejs 项目，自然是通过 npm 安装了：

```
sudo npm install -g juttle
sudo npm install -g juttle-engine
```

注意，如果是在 MacBook 上安装的话，一定要先通过 AppStore 安装好 Xcode 并确认完 license。npm 安装依赖的 sqlite3 的时候没有 xcode 会僵死在那。

juttle 包提供了命令行交互，juttle-engine 包提供了网页访问的服务器。

juttle 的配置文件默认读取位置是 `$HOME/.juttle/config.json` 。比如读取本机 elasticsearch 的数据，那么定义如下：

```
{
    "adapters": {
        "elastic": {
            "address": "localhost",
            "port": 9200
        }
    }
}
```

甚至可以读取多个不同来源的 elasticsearch，这样：

```
{
    "adapters": {
        "elastic": [{
            "id": "one",
            "address": "localhost",
            "port": 9200
        }, {
            "id": "two",
            "address": "localhost",
            "port": 9201
        }],
        "influx": {
            "url": "http://examples_influxdb_1:8086",
            "user": "root",
            "password": "root"
        }
    }
}
```

命令行运行示例

配置完成，就可以交互式命令行运行了。终端输入 `juttle` 回车进入交互界面。我们输入下面一段查询：

```
juttle> read elastic -id one -index 'logstash-*'  -from :1 year
ago: -to :now: 'MacBook-Pro' | reduce -every :1h: c = count() by
 path | filter c > 1000 | put line = 10000 | view table -columnO
rder 'time', 'c', 'line', 'path'
```

输出如下：

```
┌────────────────────────────────────┬──────────┬──────────┬───
─────────────────────────┐
│ time                               │ c        │ line     │ pat
h                        │
├────────────────────────────────────┼──────────┼──────────┼───
─────────────────────────┤
│ 2016-03-02T10:00:00.000Z           │ 4392     │ 10000    │ /va
r/log/system.log         │
├────────────────────────────────────┼──────────┼──────────┼───
─────────────────────────┤
│ 2016-03-02T11:00:00.000Z           │ 4818     │ 10000    │ /va
r/log/system.log         │
├────────────────────────────────────┼──────────┼──────────┼───
─────────────────────────┤
│ 2016-03-02T12:00:00.000Z           │ 2038     │ 10000    │ /va
r/log/system.log         │
├────────────────────────────────────┼──────────┼──────────┼───
─────────────────────────┤
│ 2016-03-02T13:00:00.000Z           │ 1826     │ 10000    │ /va
r/log/system.log         │
├────────────────────────────────────┼──────────┼──────────┼───
─────────────────────────┤
│ 2016-03-02T15:00:00.000Z           │ 10267    │ 10000    │ /va
r/log/system.log         │
├────────────────────────────────────┼──────────┼──────────┼───
─────────────────────────┤
│ 2016-03-02T16:00:00.000Z           │ 10999    │ 10000    │ /va
r/log/system.log         │
├────────────────────────────────────┼──────────┼──────────┼───
─────────────────────────┤
│ 2016-03-02T17:00:00.000Z           │ 3528     │ 10000    │ /va
r/log/system.log         │
```

```
├─────────────────────────────────────┼──────────┼──────────┼────
──────────────────────────┤
│ 2016-03-03T00:00:00.000Z          │ 2498      │ 10000     │ /va
r/log/system.log         │
├─────────────────────────────────────┼──────────┼──────────┼────
──────────────────────────┤
│ 2016-03-03T03:00:00.000Z          │ 4600      │ 10000     │ /va
r/log/system.log         │
├─────────────────────────────────────┼──────────┼──────────┼────
──────────────────────────┤
│ 2016-03-03T04:00:00.000Z          │ 7751      │ 10000     │ /va
r/log/system.log         │
├─────────────────────────────────────┼──────────┼──────────┼────
──────────────────────────┤
│ 2016-03-03T05:00:00.000Z          │ 3249      │ 10000     │ /va
r/log/system.log         │
├─────────────────────────────────────┼──────────┼──────────┼────
──────────────────────────┤
│ 2016-03-03T06:00:00.000Z          │ 5715      │ 10000     │ /va
r/log/system.log         │
├─────────────────────────────────────┼──────────┼──────────┼────
──────────────────────────┤
│ 2016-03-03T07:00:00.000Z          │ 4374      │ 10000     │ /va
r/log/system.log         │
├─────────────────────────────────────┼──────────┼──────────┼────
──────────────────────────┤
│ 2016-03-03T08:00:00.000Z          │ 2600      │ 10000     │ /va
r/log/system.log         │
└─────────────────────────────────────┴──────────┴──────────┴────
──────────────────────────┘
```

漂亮的终端表格！

警告

需要注意的是，juttle 和 es-hadoop 一样，也是通过 RESTful API 和 elasticsearch 交互，所以除了个别已经提前实现好了的 reduce 方法可以转换成 aggregation 以外，其他的 juttle 指令，都是通过 query 把数据拿回来以后，由 juttle 本身做的运算处理。juttle-adapter-elastic 模块的 `DEFAULT_FETCH_SIZE` 设置是 10000 条。

而比 es-hadoop 更差的是，因为 juttle 是单机程序，它还没有像 es-hadoop 那样并发 partition 直连每个 elasticsearch 的 shard 做并发请求。

juttle-viz 可视化界面

上一小节介绍了一下怎么用 juttle 交互式命令行查看表格式输出。juttle 事实上还提供了一个 web 服务器，做数据可视化效果，这个同样是用 juttle 语言描述配置。

我们已经安装好了 `juttle-engine` 模块，那么直接启动服务器即可：

```
~$ juttle-engine -d
```

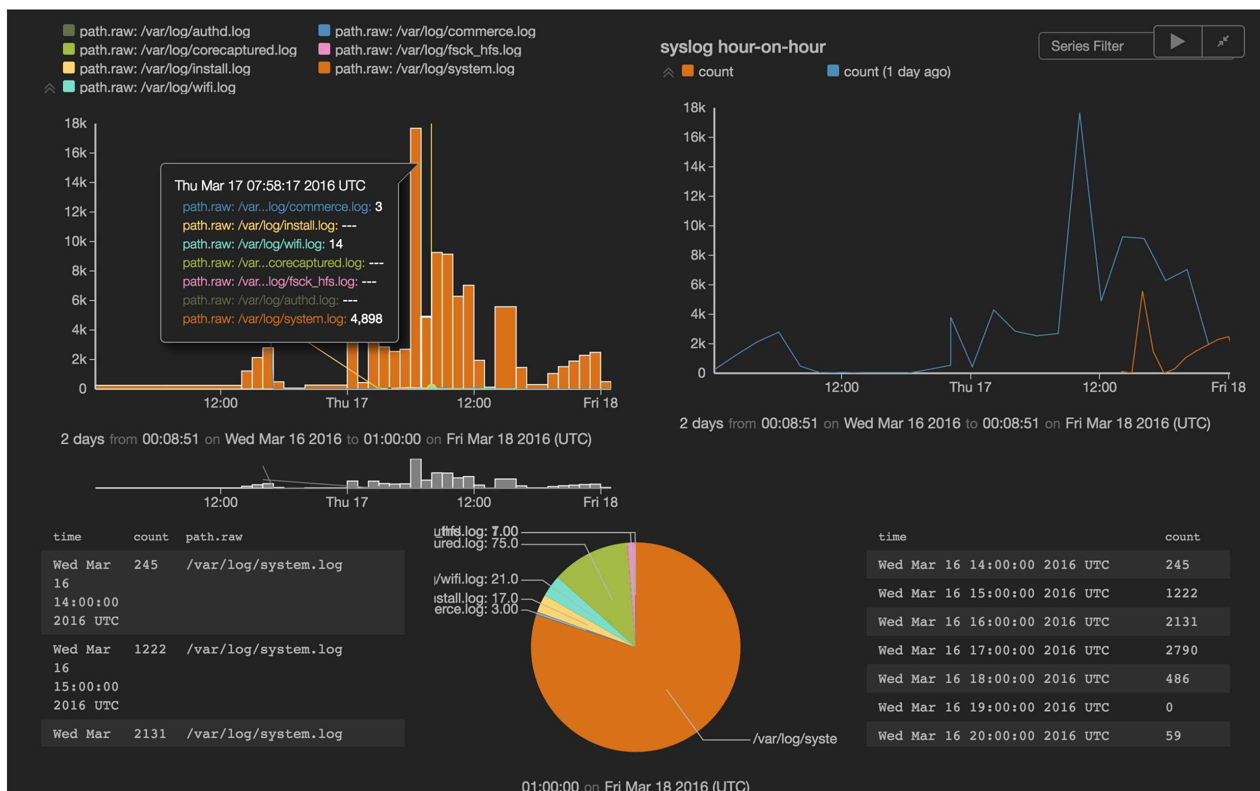
然后浏览器打开 `http://localhost:8080` 就能看到页面了。注意，请使用 Chrome v45 以上版本或者 Safari 等其他浏览器，否则有个 Array 上的 bug。

但是目前这个页面上本身不提供输入框直接写 juttle 语言。所以需要我们把 juttle 语言写成脚本文件，再来通过页面加载。

```
~$ cat > ~/test.juttle <<EOF
read elastic -index 'logstash-*'  -from :-2d: -to :now: 'MacBook
-Pro'
  | reduce -every :1h: count() by 'path.raw'
  | (
      view timechart -row 0 -col 0;;
      view table -height 200 -row 1 -col 0;
      view piechart -row 1 -col 0;
  );
(
  read elastic -index 'logstash-*'  -from :-2d: -to :-1d: 'MacBo
ok-Pro' AND '/var/log/system.log'
    | reduce -every :1h: count();
  read elastic -index 'logstash-*'  -from :-1d: -to :now: 'MacBo
ok-Pro' AND '/var/log/system.log'
    | reduce -every :1h: count();
)
  | (
      view timechart -duration :1 day: -overlayTime true -height
 400 -row 0 -col 1 -title 'syslog hour-on-hour';
      view table -height 200 -row 1 -col 1;
  );
EOF
```

然后访问 `http://localhost:8080?path=/test.juttle` ，注意这里的path参数的写法，这个/其实指的是你运行 `juttle-engine` 命令的时候的路径，而不是真的设备根目录。

就可以在浏览器上看到如下效果：



页面上还有一行有关 `path.raw` 的 WARNING 提示，那是因为 juttle 目前对 elasticsearch 的 mapping 解析支持的不是很好，但是不影响使用，可以不用管。

可视化相关指令介绍

我们可以看到这次的 juttle 脚本，跟昨天在命令行下运行的几个区别：

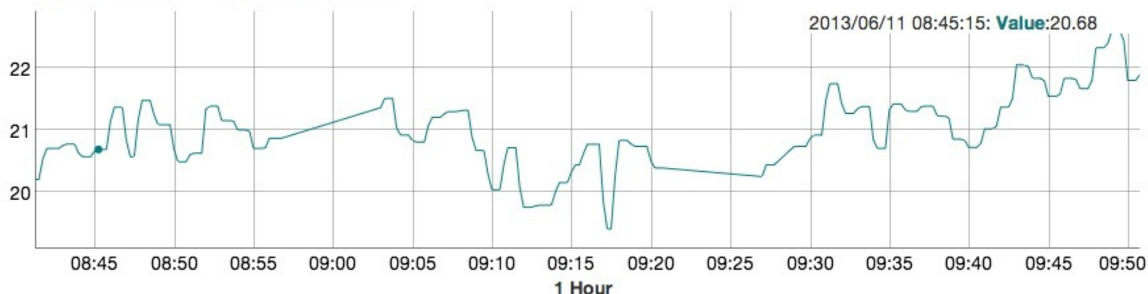
1. 我们用上了 `()` ，这是 juttle 的一大特技，对同一结果并联多个 view ，或者并联多个输入结果做相同的后续处理等等。
2. 我们对 view 用上了 `row` 和 `col` 参数，用来指定他们在页面上的布局。
3. 有一个 `timechart` 我们用了 `-durat :1d: -overlayTime true` 参数。这是 `timechart` 独有的参数，专门用来实现同比环比的。在图上的效果大家也可以看到了。不过目前也有小问题，就是鼠标放到图上的时候，只能看到第二个结果的指标说明，看不到第一个的。

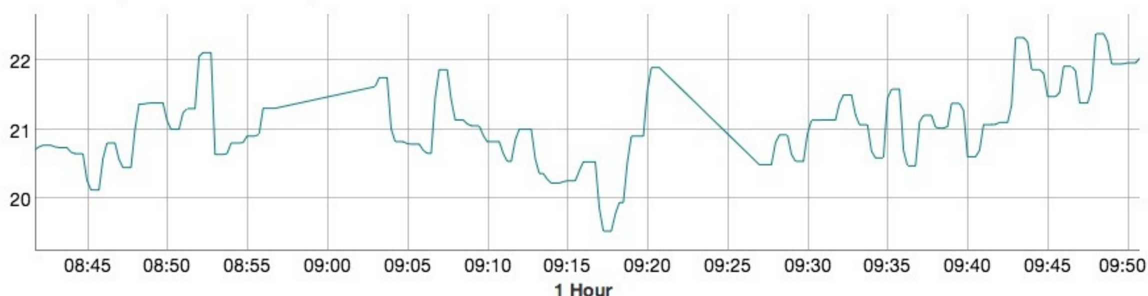
Kale 系统

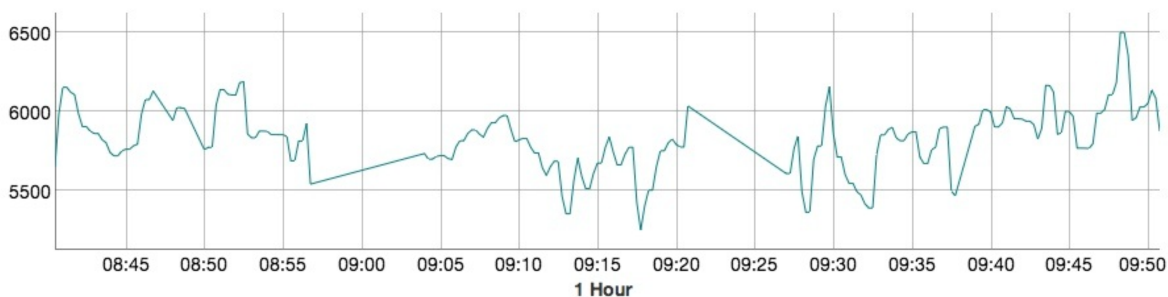
Kale 系统是 Etsy 公司开源的一个监控分析系统。Kale 分为两个部分：skyline 和 oculus。skyline 负责对时序数据进行概率分布校验，对校验失败率超过阈值的时序数据发报警；oculus 负责给被报警的时序，找出趋势相似的其他时序作为关联性参考。

看到“相似”两个字，你一定想到了。没错，oculus 组件，就是利用了 Elasticsearch 的相似度打分。

oculus 中，为 Elasticsearch 的 `org.elasticsearch.script.ExecutableScript` 扩展了 `DTW` 和 `Euclidian` 两种 NativeScript。可以在界面上选择用其中某一种算法来做相似度打分：







然后相似度最高的几个时序图就依次排列出来了。

Euclidian 即欧几里得距离，是时序相似度计算里最基础的方式。

DWT 即动态时间规整(Dynamic Time Warping)，也是时序相似度计算的常用方式，它和欧几里得距离的差别在于，欧几里得距离要求比对的时序数据是一一对应的，而动态时间规整计算的时序数据并不要求长度相等。在运维监控来说，也就是延后一定时间发生的相近趋势也可以以很高的打分项排名靠前。

不过，oculus 插件仅更新到支持 Elasticsearch-0.90.3 版本为止。Etsy 性能优化团队在 Oreilly 2015 大会上透露，他们内部已经根据 Kale 的经验教训，重新开发了 Kale 2.0 版。会在年内开源放出来。大家一起期待吧！

参考阅读

http://codeascraft.com/2013/06/11/introducing-kale/

简介

Logstash 早期曾经自带了一个特别简单的 logstash-web 用来查看 ES 中的数据。其功能太过简单，于是 Rashid Khan 用 PHP 写了一个更好用的 web，取名叫 Kibana。这个 PHP 版本的 Kibana 发布时间是 2011 年 12 月 11 日。

Kibana 迅速流行起来，不久的 2012 年 8 月 19 日，Rashid Khan 用 Ruby 重写了 Kibana，也被叫做 Kibana2。因为 Logstash 也是用 Ruby 写的，这样 Kibana 就可以替代原先那个简陋的 logstash-web 页面了。

目前我们看到的 angularjs 版本 kibana 其实原名叫 elasticsearch-dashboard，但跟 Kibana2 作者是同一个人，换句话说，kibana 比 logstash 还早就进了 elasticsearch 名下。这个项目改名 Kibana 是在 2014 年 2 月，也被叫做 Kibana3。全新的设计一下子风靡 DevOps 界。随后其他社区纷纷借鉴，Graphite 目前最流行的 Grafana 界面就是由此而来，至今代码中还留存有十余处 kbn 字样。

2014 年 4 月，Kibana3 停止开发，ES 公司集中人力开始 Kibana4 的重构，在 2015 年初发布了使用 JRuby 做后端的 beta 版后，于 3 月正式推出使用 node.js 做后端的正式版。由于设计思路上的差别，一些 K3 适宜的场景并不在 K4 考虑范围内，所以，至今 K3 和 K4 并存使用。本书也会分别讲解两者。

注释

本章配置参数内容部分译自Elasticsearch 官方指南 Kibana 部分，其中 v3 的 panel 部分额外添加了截图注释。然后在使用的基础上，添加了原创的源码解析，场景示例，二次开发入门等内容。

安排、配置和运行

Kibana4 安装方式依然简单，你可以在几分钟内安装好 Kibana 然后开始探索你的 Elasticsearch 索引。只需要预备：

- Elasticsearch 1.4.4 或者更新的版本
- 一个现代浏览器 - 支持的浏览器列表.
- 有关你的 Elasticsearch 集群的信息：
    - 你想要连接 Elasticsearch 实例的 URL
    - 你想搜索哪些 Elasticsearch 索引

> 如果你的 Elasticsearch 是被 Shield 保护着的，阅读生产环境部署章节相关内容学习额外的安装说明。

安装并启动 kibana

要安装启动 Kibana:

1. 下载对应平台的 Kibana 4 二进制包
2. 解压 `.zip` 或 `tar.gz` 压缩文件
3. 在安装目录里运行: `bin/kibana` (Linux/MacOSX) 或 `bin\kibana.bat` (Windows)

完毕！Kibana 现在运行在 5601 端口了。

让 kibana 连接到 elasticsearch

在开始用 Kibana 之前，你需要告诉它你打算探索哪个 Elasticsearch 索引。第一次访问 Kibana 的时候，你会被要求定义一个 *index pattern* 用来匹配一个或者多个索引名。好了。这就是你需要做的全部工作。以后你还可以随时从 Settings 标签页添加更多的 index pattern。

> 默认情况下，Kibana 会连接运行在 `localhost` 的 Elasticsearch。要连接其他 Elasticsearch 实例，修改 `kibana.yml` 里的 Elasticsearch URL，然后重启 Kibana。如何在生产环境下使用 Kibana，阅读生产环境部署章节。

要从 Kibana 访问的 Elasticsearch 索引的配置方法：

1. 从浏览器访问 Kibana 界面。也就是说访问比如 `localhost:5601` 或者 `http://YOURDOMAIN.com:5601` 。

   Configure an index pattern

   In order to use Kibana you must configure at least one index pattern. Index patterns are used to identify the Elasticsearch index to run search and analytics against. They are also used to configure fields.

   Index contains time-based events

   Use event times to create index names [DEPRECATED]

   Index name or pattern

   Patterns allow you to define dynamic index names using * as a wildcard. Example: logstash-*

   logstash-*

   Do not expand index pattern when searching (Not recommended)

   By default, searches against any time-based index pattern that contains a wildcard will automatically be expanded to query only the indices that contain data within the currently selected time range.

   Searching against the index pattern *logstash-** will actually query elasticsearch for the specific matching indices (e.g. *logstash-2015.12.21*) that fall within the current time range.

   Time-field name refresh fields

   Create

2. 指定一个可以匹配一个或者多个 Elasticsearch 索引的 index pattern 。默认情况下，Kibana 认为你要访问的是通过 Logstash 导入 Elasticsearch 的数据。这时候你可以用默认的 `logstash-*` 作为你的 index pattern。通配符(*) 匹配索引名中零到多个字符。如果你的 Elasticsearch 索引有其他命名约定，输入合适的 pattern。pattern 也开始是最简单的单个索引的名字。
3. 选择一个包含了时间戳的索引字段，可以用来做基于时间的处理。Kibana 会读取索引的映射，然后列出所有包含了时间戳的字段(译者注：实际是字段类型为 date 的字段，而不是"看起来像时间戳"的字段)。如果你的索引没有基于时间的数据，关闭 `Index contains time-based events` 参数。
4. 如果一个新索引是定期生成，而且索引名中带有时间戳，选择 `Use event times to create index names` 选项，然后再选择 `Index pattern interval` 。这可以提高搜索性能，Kibana 会至搜索你指定的时间范围内的索引。在你用 Logstash 输出数据给 Elasticsearch 的情况下尤其有效。
5. 点击 `Create` 添加 index pattern。第一个被添加的 pattern 会自动被设置为默认值。如果你有多个 index pattern 的时候，你可以在 `Settings > Indices` 里设置具体哪个是默认值。

好了。Kibana 现在连接上你的 Elasticsearch 数据了。Kibana 会显示匹配上的索引里的字段名的只读列表。

开始探索你的数据！

你可以开始下钻你的数据了：

- 在 Discover 页搜索和浏览你的数据。
- 在 Visualize 页转换数据成图表。
- 在 Dashboard 页创建定制自己的仪表板。

生产环境部署

Kibana5 是是一个完整的 web 应用。使用时，你需要做的只是打开浏览器，然后输入你运行 Kibana 的机器地址然后加上端口号。比如说： `localhost:5601` 或者 `http://YOURDOMAIN.com:5601` 。

但是当你准备在生产环境使用 Kibana5 的时候，比起在本机运行，就需要多考虑一些问题：

- 在哪运行 kibana
- 是否需要加密 Kibana 出入的流量
- 是否需要控制访问数据的权限

Nginx 代理配置

因为 Kibana5 不再是 Kibana3 那种纯静态文件的单页应用，所以其服务器端是需要消耗计算资源的。因此，如果用户较多，Kibana5 确实有可能需要进行多点部署，这时候，就需要用 Nginx 做一层代理了。

和 Kibana3 相比，Kibana5 的 Nginx 代理配置倒是简单许多，因为所有流量都是统一配置的。下面是一段包含入口流量加密、简单权限控制的 Kibana5 代理配置：

```
upstream kibana5 {
    server 127.0.0.1:5601 fail_timeout=0;
}
server {
    listen                *:80;
    server_name           kibana_server;
    access_log            /var/log/nginx/kibana.srv-log-dev.log;
    error_log             /var/log/nginx/kibana.srv-log-dev.error
.log;

    ssl                   on;
    ssl_certificate       /etc/nginx/ssl/all.crt;
    ssl_certificate_key   /etc/nginx/ssl/server.key;

    location / {
        root   /var/www/kibana;
        index  index.html  index.htm;
    }

    location ~ ^/kibana5/.* {
        proxy_pass           http://kibana5;
        rewrite              ^/kibana5/(.*)  /$1 break;
        proxy_set_header     X-Forwarded-For $proxy_add_x_forwar
ded_for;
        proxy_set_header     Host            $host;
        auth_basic           "Restricted";
        auth_basic_user_file /etc/nginx/conf.d/kibana.myhost.org
.htpasswd;
    }
}
```

如果用户够多，当然你可以单独跑一个 kibana5 集群，然后在 `upstream` 配置段中添加多个代理地址做负载均衡。

配置 Kibana 和 shield 一起工作

Nginx 只能加密和管理浏览器到服务器端的请求，而 Kibana5 到 ELasticsearch 集群的请求，就需要由 Elasticsearch 方面来完成了。如果你在用 Shield 做 Elasticsearch 用户认证，你需要给 Kibana 提供用户凭证，这样它才能访问 `.kibana` 索引。Kibana 用户需要由权限访问 `.kibana` 索引里以下操作：

```
'.kibana':
      - indices:admin/create
      - indices:admin/exists
      - indices:admin/mapping/put
      - indices:admin/mappings/fields/get
      - indices:admin/refresh
      - indices:admin/validate/query
      - indices:data/read/get
      - indices:data/read/mget
      - indices:data/read/search
      - indices:data/write/delete
      - indices:data/write/index
      - indices:data/write/update
      - indices:admin/create
```

更多配置 Shield 的内容，请阅读官网的 Shield with Kibana 4。

要配置 Kibana 的凭证，设置 `kibana.yml` 里的 `kibana_elasticsearch_username` 和 `kibana_elasticsearch_password` 选项即可：

```
# If your Elasticsearch is protected with basic auth:
kibana_elasticsearch_username: kibana5
kibana_elasticsearch_password: kibana5
```

开启 ssl

Kibana 同时支持对客户端请求以及 Kibana 服务器发往 Elasticsearch 的请求做 SSL 加密。

要加密浏览器(或者在 Nginx 代理情况下，Nginx 服务器)到 Kibana 服务器之间的通信，配置 `kibana.yml` 里的 `ssl_key_file` 和 `ssl_cert_file` 参数：

```
# SSL for outgoing requests from the Kibana Server (PEM formatte
d)
ssl_key_file: /path/to/your/server.key
ssl_cert_file: /path/to/your/server.crt
```

如果你在用 Shield 或者其他提供 HTTPS 的代理服务器保护 Elasticsearch，你可以配置 Kibana 通过 HTTPS 方式访问 Elasticsearch，这样 Kibana 服务器和 Elasticsearch 之间的通信也是加密的。

要做到这点，你需要在 `kibana.yml` 里配置 Elasticsearch 的 URL 时指明是 HTTPS 协议：

```
elasticsearch: "https://<your_elasticsearch_host>.com:9200"
```

如果你给 Elasticsearch 用的是自己签名的证书，请在 `kibana.yml` 里设定 `ca` 参数指明 PEM 文件位置，这也意味着开启了 `verify_ssl` 参数：

```
# If you need to provide a CA certificate for your Elasticsarech
 instance, put
# the path of the pem file here.
ca: /path/to/your/ca/cacert.pem
```

控制访问权限

你可以用 Elasticsearch Shield 来控制用户通过 Kibana 可以访问到的 Elasticsearch 数据。Shield 提供了索引级别的访问控制。如果一个用户没被许可运行这个请求，那么它在 Kibana 可视化界面上只能看到一个空白。

要配置 Kibana 使用 Shield，你要为 Kibana 创建一个或者多个 Shield 角色(role)，以 `kibana5` 作为开头的默认角色。更详细的做法，请阅读 Using Shield with Kibana 4。

discover 功能

Discover 标签页用于交互式探索你的数据。你可以访问到匹配得上你选择的索引模式的每个索引的每条记录。你可以提交搜索请求，过滤搜索结果，然后查看文档数据。你还可以看到匹配搜索请求的文档总数，获取字段值的统计情况。如果索引模式配置了时间字段，文档的时序分布情况会在页面顶部以柱状图的形式展示出来。



设置时间过滤器

时间过滤器(Time Filter)限制搜索结果在一个特定的时间周期内。如果你的索引包含的是时序诗句，而且你为所选的索引模式配置了时间字段，那么就就可以设置时间过滤器。

默认的时间过滤器设置为最近 15 分钟。你可以用页面顶部的时间选择器(Time Picker)来修改时间过滤器，或者选择一个特定的时间间隔，或者直方图的时间范围。

要用时间选择器来修改时间过滤器：

1. 点击菜单栏右上角显示的 Time Filter 打开时间选择器。
2. 快速过滤，直接选择一个短链接即可。
3. 要指定相对时间过滤，点击 Relative 然后输入一个相对的开始时间。可以是任意数字的秒、分、小时、天、月甚至年之前。
4. 要指定绝对时间过滤，点击 Absolute 然后在 From 框内输入开始日期，To 框内输入结束日期。
5. 点击时间选择器底部的箭头隐藏选择器。

要从柱状图上设置时间过滤器，有以下几种方式：

- 想要放大那个时间间隔，点击对应的柱体。
- 单击并拖拽一个时间区域。注意需要等到光标变成加号，才意味着这是一个有效的起始点。

你可以用浏览器的后退键来回退你的操作。

搜索数据

在 Discover 页提交一个搜索，你就可以搜索匹配当前索引模式的索引数据了。你可以直接输入简单的请求字符串，也就是用 Lucene query syntax，也可以用完整的基于 JSON 的 Elasticsearch Query DSL。

当你提交搜索的时候，直方图，文档表格，字段列表，都会自动反映成搜索的结果。hits(匹配的文档)总数会在直方图的右上角显示。文档表格显示前 500 个匹配文档。默认的，文档倒序排列，最新的文档最先显示。你可以通过点击时间列的头部来反转排序。事实上，所有建了索引的字段，都可以用来排序，稍后会详细说明。

要搜索你的数据：

1. 在搜索框内输入请求字符串：
    - 简单的文本搜索，直接输入文本字符串。比如，如果你在搜索网站服务器日志，你可以输入 `safari` 来搜索各字段中的 `safari` 单词。
    - 要搜索特定字段中的值，则在值前加上字段名。比如，你可以输入 `status:200` 来限制搜索结果都是在 `status` 字段里有 `200` 内容。
    - 要搜索一个值的范围，你可以用范围查询语法， `[START_VALUE TO END_VALUE]` 。比如，要查找 4xx 的状态码，你可以输入 `status:[400 TO 499]` 。
    - 要指定更复杂的搜索标准，你可以用布尔操作符 `AND` , `OR` , 和 `NOT` 。比如，要查找 4xx 的状态码，还是 `php` 或 `html` 结尾的数据，你可以输入 `status:[400 TO 499] AND (extension:php OR extension:html)` 。

> 这些例子都用了 Lucene query syntax。你也可以提交 Elasticsearch Query DSL 式的请求。更多示例，参见之前 Elasticsearch 章节。

1. 点击回车键，或者点击 `Search` 按钮提交你的搜索请求。

开始一个新的搜索

要清除当前搜索或开始一个新搜索，点击 Discover 工具栏的 New Search 按钮。

保存搜索

你可以在 Discover 页加载已保存的搜索，也可以用作 visualizations 的基础。保存一个搜索，意味着同时保存下了搜索请求字符串和当前选择的索引模式。

要保存当前搜索：

1. 点击 Discover 工具栏的 `Save Search` 按钮。
2. 输入一个名称，点击 `Save` 。

加载一个已存搜索

要加载一个已保存的搜索：

1. 点击 Discover 工具栏的 `Load Search` 按钮。
2. 选择你要加载的搜索。

如果已保存的搜索关联到跟你当前选择的索引模式不一样的其他索引上，加载这个搜索也会切换当前的已选索引模式。

改变你搜索的索引

当你提交一个搜索请求，匹配当前的已选索引模式的索引都会被搜索。当前模式模式会显示在搜索栏下方。要改变搜索的索引，需要选择另外的模式模式。

要选择另外的索引模式：

1. 点击 Discover 工具栏的 `Settings` 按钮。
2. 从索引模式列表中选取你打算采用的模式。

关于索引模式的更多细节，请阅读稍后 Setting 功能小节。

自动刷新页面

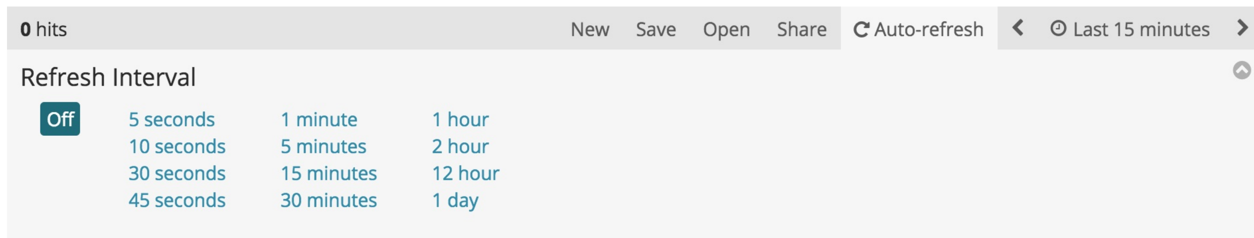
亦可以配置一个刷新间隔来自动刷新 Discover 页面的最新索引数据。这会定期重新提交一次搜索请求。

设置刷新间隔后，会显示在菜单栏时间过滤器的左边。

要设置刷新间隔：

1. 点击菜单栏右上角的 `Time Picker` 🕘。

2. 点击 `Auto refresh` 标签。
3. 从列表中选择一个刷新间隔。



开启自动刷新后，Kibana 的顶部栏会出现一个暂停按钮和自动刷新的间隔，点击 **Pause** 按钮可以暂停自动刷新。

按字段过滤

你可以过滤搜索结果，只显示在某字段中包含了特定值的文档。也可以创建反向过滤器，排除掉包含特定字段值的文档。

你可以从字段列表或者文档表格里添加过滤器。当你添加好一个过滤器后，它会显示在搜索请求下方的过滤栏里。从过滤栏里你可以编辑或者关闭一个过滤器，转换过滤器(从正向改成反向，反之亦然)，切换过滤器开关，或者完全移除掉它。

要从字段列表添加过滤器：

1. 点击你想要过滤的字段名。会显示这个字段的前 5 名数据。每个数据的右侧，有两个小按钮 —— 一个用来添加常规(正向)过滤器，一个用来添加反向过滤器。
2. 要添加正向过滤器，点击 `Positive Filter` 按钮 。这个会过滤掉在本字段不包含这个数据的文档。
3. 要添加反向过滤器，点击 `Negative Filter` 按钮 。这个会过滤掉在本字段包含这个数据的文档。

要从文档表格添加过滤器：

1. 点击表格第一列(通常都是时间)文档内容左侧的 `Expand` 按钮 展开文档表格中的文档。每个字段名的右侧，有两个小按钮 —— 一个用来添加常规(正向)过滤器，一个用来添加反向过滤器。
2. 要添加正向过滤器，点击 `Positive Filter` 按钮 。这个会过滤掉在本字段不包含这个数据的文档。
3. 要添加反向过滤器，点击 `Negative Filter` 按钮 。这个会过滤掉在本字

段包含这个数据的文档。

过滤器(Filter)的协同工作方式

在 Kibana 的任意页面创建过滤器后，就会在搜索输入框的下方，出现椭圆形的过滤条件。

鼠标移动到过滤条件上，会显示下面几个图标：



- 过滤器开关 点击这个图标，可以在不移除过滤器的情况下关闭过滤条件。再次点击则重新打开。被禁用的过滤器是条纹状的灰色，要求包含(相当于 Kibana3 里的 must)的过滤条件显示为绿色，要求排除(相当于 Kibana3 里的 mustNot)的过滤条件显示为红色。
- 过滤器图钉 点击这个图标钉住过滤器。被钉住的过滤器，可以横贯 Kibana 各个标签生效。比如在 *Visualize* 标签页钉住一个过滤器，然后切换到 *Discover* 或者 *Dashboard* 标签页，过滤器依然还在。注意：如果你钉住了过滤器，然后发现检索结果为空，注意查看当前标签页的索引模式是不是跟过滤器匹配。
- 过滤器反转 点击这个图标反转过滤器。默认情况下，过滤器都是包含型，显示为绿色，只有匹配过滤条件的结果才会显示。反转成排除型过滤器后，显示的是不匹配过滤器的检索项，显示为红色。
- 移除过滤器 点击这个图标删除过滤器。
- 自定义过滤器 点击这个图标会打开一个文本编辑框。编辑框内可以修改 JSON 形式的过滤器内容，并起一个 alias 别名： JSON 中可以灵活应用 bool query 组合各种 `should` 、 `must` 、 `must_not` 条件。一个用 `should` 表达的 OR 关系过滤如下:

```
{
  "bool": {
      "should": [
          {
              "term": {
                  "geoip.country_name.raw": "Canada"
              }
          },
          {
              "term": {
                  "geoip.country_name.raw": "China"
              }
          }
      ]
  }
}
```

想要对当前页所有过滤器统一执行上面的某个操作，点击 **Actions** 按钮，然后再执行操作即可。

查看文档数据

当你提交一个搜索请求，最近的 500 个搜索结果会显示在文档表格里。你可以在 **Advanced Settings** 里通过 `discover:sampleSize` 属性配置表格里具体的文档数量。默认的，表格会显示当前选择的索引模式中定义的时间字段内容(转换成本地时区)以及 `_source` 文档。你可以从字段列表添加字段到文档表格。还可以用表格里包含的任意已建索引的字段来排序列出的文档。

要查看一个文档的字段数据，点击表格第一列(通常都是时间)文档内容左侧的 `Expand` 按钮 ▶ 。Kibana 从 Elasticsearch 读取数据然后在表格中显示文档字段。这个表格每行是一个字段的名字、过滤器按钮和字段的值。

Time	_source
April 13th 2017, 22:44:03.388	**index:** logstash-2017.04.13 **@timestamp:** April 13th 2017, 22:44:03.388 **ip:** 107.122.180.120 **extension:** jpg **response:** 200 **geo.coordinates:** { "lat": 39.90415167, "lon": -74.74954917 } **geo.src:** MY **geo.dest:** US **geo.srcdest:** MY:US **@tags:** success, info **utc_time:** April 13th 2017, 22:44:03.388 **referer:** http://twitter.com/warning/robert-satcher **agent:** Mozilla/5.0 (X11; Linux i686) AppleWebKit/534.24 (KHTML, like Gec

Table JSON View surrounding documents View single document

t @message		107.122.180.120 - - [2017-04-13T20:44:03.388Z] "GET /uploads/shannon-walker.jpg HTTP/1.1" 200 8887 "-" "Mozilla/5.0 (X11; Linux i686) AppleWebKit/534.24 (KHTML, like Gecko) Chrome/11.0.696.50 Safari/534.24"
t @tags		success, info

1. 要查看原始 JSON 文档(格式美化过的)，点击 **JSON** 标签。
2. 要在单独的页面上查看文档内容，点击链接。你可以添加书签或者分享这个链接，以直接访问这条特定文档。
3. 收回文档细节，点击 **Collapse** 按钮 。
4. To toggle a particular field's column in the Documents table, click the **Toggle column in table** button.

文档列表排序

你可以用任意已建索引的字段排序文档表格中的数据。如果当前索引模式配置了时间字段，默认会使用该字段倒序排列文档。

要改变排序方式：

- 点击想要用来排序的字段名。能用来排序的字段在字段名右侧都有一个排序按钮。再次点击字段名，就会反向调整排序方式。

给文档表格添加字段列

默认的，文档表格会显示当前选择的索引模式中定义的时间字段内容(转换成本地时区)以及 `_source` 文档。你可以从字段列表添加字段到文档表格。

要添加字段列到文档表格：

1. 从字段列表中添加一列，移动鼠标到字段列表的字段上，点击它的 `add` 按钮
2. 从文档的字段数据添加一列，展开对应的文档后点击字段的 `Toggle column in table` 按钮

添加的字段会替换掉文档表格里的 `_source` 列。同时还会显示在字段列表顶部的 `Selected Fields` 区域里。

要重排表格中的字段列，移动鼠标到你要移动的列顶部，点击移动过按钮。

	Time	geo.src	geo.dest	response
▾	October 21st 2016, 16:36:00.038	VN	IN	404

Move column to the left

从文档表格删除字段列

要从文档表格删除字段列：

1. 移动鼠标到字段列表的 `Selected Fields` 区域里你想要移除的字段上，然后点击它的 `remove` 按钮 ✖。
2. 重复操作直到你移除完所有你想移除的字段。

查看字段数据统计

从字段列表，你可以看到文档表格里有多少数据包含了这个字段，排名前 5 的值是什么，以及包含各个值的文档的占比。

要查看字段数据统计，在字段列表中点击对应的字段名即可。



> 要基于这个字段创建可视化，点击字段统计下方的 Visualize 按钮。

各 Visualize 功能

Visualize 标签页用来设计可视化。你可以保存可视化，以后再用，或者加载合并到 *dashboard* 里。一个可视化可以基于以下几种数据源类型：

- 一个新的交互式搜索
- 一个已保存的搜索
- 一个已保存的可视化

可视化是基于 Elasticsearch 1.0 引入的聚合(aggregation) 特性。

创建一个新可视化

要开始一个 **Create New Visualization** 向导，点击页面左侧边栏的 **Visualize** 标签。如果你已经在浏览一个可视化了，你可以在顶部菜单栏里点击 **New** 选项！向导会引导你继续以下几步：

第 1 步: 选择可视化类型

在 **New Visualization** 向导起始页可以选择以下一个可视化类型：

类型	用途
Area chart	用区块图来可视化多个不同序列的总体贡献。
Data table	用数据表来显示聚合的原始数据。其他可视化可以通过点击底部的方式显示数据表。
Line chart	用折线图来比较不同序列。
Markdown widget	用 Markdown 显示自定义格式的信息或和你仪表盘有关的用法说明。
Metric	用指标可视化在你仪表盘上显示单个数字。
Pie chart	用饼图来显示每个来源对总体的贡献。
Tile map	用瓦片地图将聚合结果和经纬度联系起来。
Timeseries	计算和展示多个时间序列数据。
Vertical bar chart	用垂直条形图作为一个通用图形。

你也可以加载一个你之前创建好保存下来的可视化。已存可视化选择器包括一个文本框用来过滤可视化名称，以及一个指向 对象编辑器**(Object Editor)** 的链接，可以通过 **Settings > Edit Saved Objects** 来管理已存的可视化。

如果你的新可视化是一个 **Markdown** 或 **Timeseries** 挂件，会直接进入配置界面。其他的可视化类型，选择后都会转到数据源选择。

第 2 步: 选择数据源

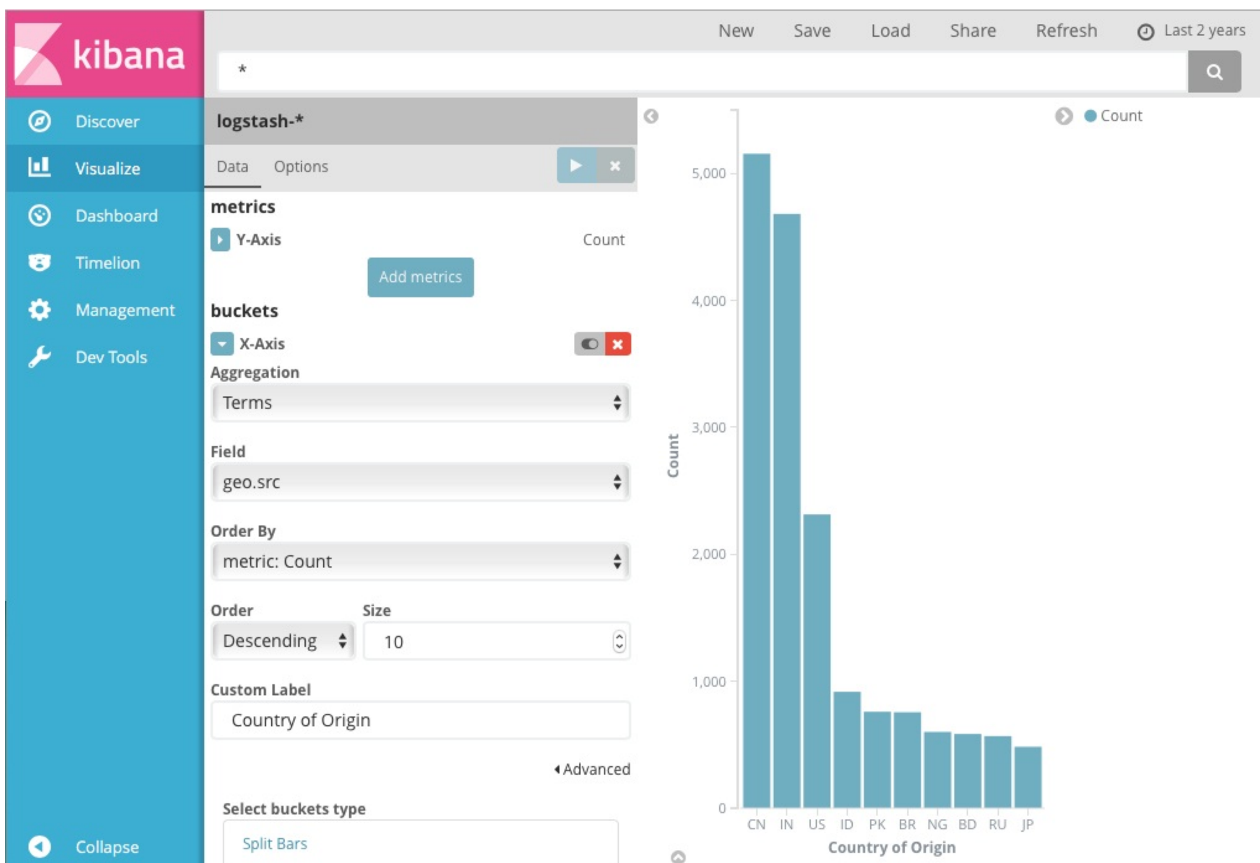
你可以选择新建或者读取一个已保存的搜索，作为你可视化的数据源。搜索是和一个或者一系列索引相关联的。如果你选择了在一个配置了多个索引的系统上开始你的新搜索，从可视化编辑器的下拉菜单里选择一个索引模式。

当你从一个已保存的搜索开始创建并保存好了可视化，这个搜索就绑定在这个可视化上。如果你修改了搜索，对应的可视化也会自动更新。

第 3 步: 可视化编辑器

可视化编辑器用来配置编辑可视化。它有下面几个主要元素：

1. 工具栏(Toolbar)
2. 聚合构建器(Aggregation Builder)
3. 预览画布(Preview Canvas)



工具栏

工具栏上有一个用户交互式数据搜索的搜索框，用来保存和加载可视化。因为可视化是基于保存好的搜索，搜索栏会变成灰色。要编辑搜索，双击搜索框，用编辑后的版本替换已保存搜索。

搜索框右侧的工具栏有一系列按钮，用于创建新可视化，保存当前可视化，加载一个已有可视化，分享或内嵌可视化，和刷新当前可视化的数据。

聚合构建器

用页面左侧的聚合构建器配置你的可视化要用的 metric 和 bucket 聚合。桶 (Buckets) 的效果类似于 SQL `GROUP BY` 语句。想更详细的了解聚合，阅读 Elasticsearch aggregations reference。

在条带图或者折线图可视化里，用 *metrics* 做 Y 轴，然后 *buckets* 做 X 轴，条带颜色，以及行/列的区分。在饼图里，*metrics* 用来做分片的大小，*buckets* 做分片的数量。

为你的可视化 Y 轴选一个 metric 聚合，包括 count, average, sum, min, max, or cardinality (unique count). 为你的可视化 X 轴，条带颜色，以及行/列的区分选一个 bucket 聚合，常见的有 date histogram, range, terms, filters, 和 significant terms。

你可以设置 buckets 执行的顺序。在 Elasticsearch 里，第一个聚合决定了后续聚合的数据集。下面例子演示一个网页访问量前五名的文件后缀名统计的时间条带图。

要看所有相同后缀名的，设置顺序如下：

1. `Color` : 后缀名的 Terms 聚合
2. `X-Axis` : `@timestamp` 的时间条带图

Elasticsearch 收集记录，算出前 5 名后缀名，然后为每个后缀名创建一个时间条带图。

要看每个小时的前 5 名后缀名情况，设置顺序如下：

1. `X-Axis` : `@timestamp` 的时间条带图(1 小时间隔)
2. `Color` : 后缀名的 Terms 聚合

这次，Elasticsearch 会从所有记录里创建一个时间条带图，然后在每个桶内，分组(本例中就是一个小时的间隔)计算出前 5 名的后缀名。

> 记住，每个后续的桶，都是从前一个的桶里分割数据。

要在预览画布 *(preview canvas)* 上渲染可视化，点击聚合构建器底部的 **Apply** 按钮。

预览画布(canvas)

预览 canvas 上显示你定义在聚合构建器里的可视化的预览效果。要刷新可视化预览，点击工具栏里的 **Refresh**。

区块图

这个图的 Y 轴是数值维度。该维度有以下聚合可用：

- Count count 聚合返回选中索引模式中元素的原始计数。
- Average 这个聚合返回一个数值字段的 average 。从下拉菜单选择一个字段。
- Sum sum 聚合返回一个数值字段的总和。从下拉菜单选择一个字段。
- Min min 聚合返回一个数值字段的最小值。从下拉菜单选择一个字段。
- Max max 聚合返回一个数值字段的最大值。从下拉菜单选择一个字段。
- Unique Count cardinality 聚合返回一个字段的去重数据值。从下拉菜单选择一个字段。
- Standard Deviation extended stats 聚合返回一个数值字段数据的标准差。从下拉菜单选择一个字段。
- Percentile percentile 聚合返回一个数值字段中值的百分比分布。从下拉菜单选择一个字段，然后在 **Percentiles** 框内指定范围。点击 **X** 移除一个百分比框，点击 **+Add** 添加一个百分比框。
- Percentile Rank percentile ranks 聚合返回一个数值字段中你指定值的百分位排名。从下拉菜单选择一个字段，然后在 **Values** 框内指定一到多个百分位排名值。点击 **X** 移除一个百分比框，点击 **+Add** 添加一个数值框。

你可以点击 **+ Add Aggregation** 按键添加一个聚合。

buckets 聚合指明从你的数据集中将要检索什么信息。

图形的 X 轴是*buckets* 维度。你可以为 X 轴定义 buckets，同样还可以为图片上的分片区域，或者分割的图片定义 buckets。

该图形的 X 轴支持以下聚合。点击每个聚合的链接查看该聚合的 Elasticsearch 官方文档。

- Date Histogram date histogram 基于数值字段创建，由时间组织起来。你可以指定时间片的间隔，单位包括秒，分，小时，天，星期，月，年。
- Histogram 标准 histogram 基于数值字段创建。为这个字段指定一个整数间隔。勾选 **Show empty buckets** 让直方图中包含空的间隔。
- Range 通过 range 聚合。你可以为一个数值字段指定一系列区间。点击 **Add Range** 添加一对区间端点。点击红色 **(x)** 符号移除一个区间。
- Date Range date range 聚合计算你指定的时间区间内的值。你可以使用 date

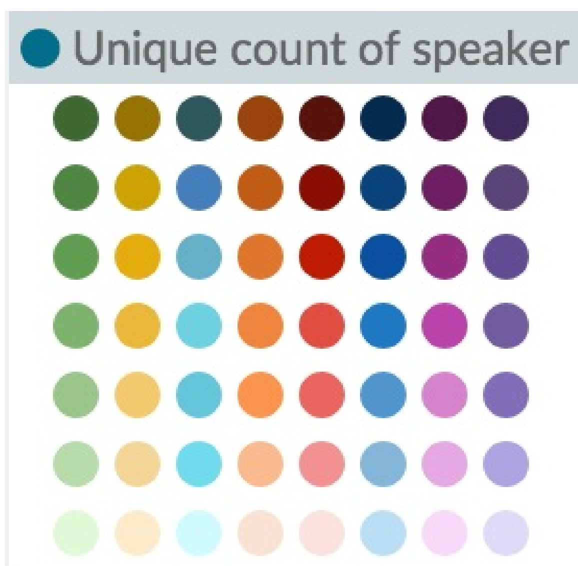
math 表达式指定区间。点击 **Add Range** 添加新的区间端点。点击红色 **(/)** 符号移除区间。

- IPv4 Range IPv4 range 聚合用来指定 IPv4 地址的区间。点击 **Add Range** 添加新的区间端点。点击红色 **(/)** 符号移除区间。
- Terms terms 聚合允许你指定展示一个字段的首尾几个元素，排序方式可以是计数或者其他自定义的metric。
- Filters 你可以为数据指定一组 filters。你可以用 query string，也可以用 JSON 格式来指定过滤器，就像在 Discover 页的搜索栏里一样。点击 **Add Filter** 添加下一个过滤器。
- Significant Terms 展示实验性的 significant terms 聚合的结果。

一旦你定义好了一个 X 轴聚合。你可以继续定义子聚合来完善可视化效果。点击 **+ Add Sub Aggregation** 添加子聚合，然后选择 **Split Area** 或者 **Split Chart**，然后从类型菜单中选择一个子聚合。

当一个图形中定义了多个聚合，你可以使用聚合类型右侧的上下箭头来改变聚合的优先级。比如，一个事件计数的日期图，可以按照时序显示，你也可以提升事件聚合的优先级，首先显示最活跃的几天。时序图用来显示事件随着时间变化的趋势，而按照活跃时间排序则可以揭示你数据中的部分异常值。

Kibana 4.5 以后新增了两个呼唤已久的功能：在 **Custom Label** 里填写自定义字符串，就可以修改显示的标签文字。而在具体标签值旁边的颜色上点击，可以打开颜色选择器，自定义自己的可视化效果的颜色。



你可以点击 **Advanced** 链接显示更多有关聚合的自定义参数：

- Exclude Pattern 指定一个从结果集中排除掉的模式。

- Exclude Pattern Flags 排除模式的 Java flags 标准集。
- Include Pattern 指定一个从结果集中要包含的模式。
- Include Pattern Flags 包含模式的 Java flags 标准集。
- JSON Input 一个用来添加 JSON 格式属性的文本框，内容会合并进聚合的定义中，格式如下例：

```
{ "script" : "doc['grade'].value * 1.2" }
```

注意

> Elasticsearch 1.4.3 及以后版本，这个函数需要你开启 dynamic Groovy scripting。

这些参数是否可用，依赖于你选择的聚合函数。

选择 **view options** 更改表格中如下方面：

- Chart Mode 当你为图形定义了多个 Y 轴时，你可以用该下拉菜单定义聚合如何显示在图形上：
    - stacked 聚合结果依次叠加在顶部。
    - overlap 聚合结果重叠的地方采用半透明效果。
    - wiggle 聚合结果显示成 streamgraph 效果。
    - percentage 显示每个聚合在总数中的百分值。
    - silhouette 显示每个聚合距离中间线的方差。

多选框可以用来控制以下行为：

- Smooth Lines 勾选该项平滑数据点之间的折线成曲线。
- Set Y-Axis Extents 勾选该项，然后在 **y-max** 和 **y-min** 框里输入数值限定 Y 轴为指定数值。
- Scale Y-Axis to Data Bounds 默认的 Y 轴长度为 0 到数据集的最大值。勾选该项改变 Y 轴的最大和最小值为数据集的返回值。
- Show Tooltip 勾选该项显示工具栏。
- Show Legend 勾选该项在图形右侧显示图例。

数据表格

- Count count 聚合返回选中索引模式中元素的原始计数。
- Average 这个聚合返回一个数值字段的 average 。从下拉菜单选择一个字段。
- Sum sum 聚合返回一个数值字段的总和。从下拉菜单选择一个字段。
- Min min 聚合返回一个数值字段的最小值。从下拉菜单选择一个字段。
- Max max 聚合返回一个数值字段的最大值。从下拉菜单选择一个字段。
- Unique Count cardinality 聚合返回一个字段的去重数据值。从下拉菜单选择一个字段。
- Standard Deviation extended stats 聚合返回一个数值字段数据的标准差。从下拉菜单选择一个字段。
- Percentile percentile 聚合返回一个数值字段中值的百分比分布。从下拉菜单选择一个字段，然后在 **Percentiles** 框内指定范围。点击 **X** 移除一个百分比框，点击 **+ Add Percent** 添加一个百分比框。
- Percentile Rank percentile ranks 聚合返回一个数值字段中你指定值的百分位排名。从下拉菜单选择一个字段，然后在 **Values** 框内指定一到多个百分位排名值。点击 **X** 移除一个百分比框，点击 **+Add** 添加一个数值框。

你可以点击 **+ Add Aggregation** 按键添加一个聚合。

数据表格的每行，叫做 *buckets*。你可以定义 buckets 来切割表格成行，或者切割表格成另一个表格。

每个 bucket 类型都支持以下聚合：

- Date Histogram date histogram 基于数值字段创建，由时间组织起来。你可以指定时间片的间隔，单位包括秒，分，小时，天，星期，月，年。
- Histogram 标准 histogram 基于数值字段创建。为这个字段指定一个整数间隔。勾选 **Show empty buckets** 让直方图中包含空的间隔。
- Range 通过 range 聚合。你可以为一个数值字段指定一系列区间。点击 **Add Range** 添加一堆区间端点。点击红色 **(x)** 符号移除一个区间。
- Date Range date range 聚合计算你指定的时间区间内的值。你可以使用 date math 表达式指定区间。点击 **Add Range** 添加新的区间端点。点击红色 **(/)** 符号移除区间。
- IPv4 Range IPv4 range 聚合用来指定 IPv4 地址的区间。点击 **Add Range** 添加新的区间端点。点击红色 **(/)** 符号移除区间。

- Terms terms 聚合允许你指定展示一个字段的首尾几个元素，排序方式可以是计数或者其他自定义的metric。
- Filters 你可以为数据指定一组 filters。你可以用 query string，也可以用 JSON 格式来指定过滤器，就像在 Discover 页的搜索栏里一样。点击 **Add Filter** 添加下一个过滤器。
- Significant Terms 展示实验性的 significant terms 聚合的结果。
- Geohash geohash 聚合显示基于地理坐标的点。

一旦你定义好了一个 X 轴聚合。你可以继续定义子聚合来完善可视化效果。点击 **+ Add Sub Aggregation** 添加子聚合，然后选择 **Split Area** 或者 **Split Chart**，然后从类型菜单中选择一个子聚合。

当一个图形中定义了多个聚合，你可以使用聚合类型右侧的上下箭头来改变聚合的优先级。

你可以点击 **Advanced** 链接显示更多有关聚合的自定义参数：

- Exclude Pattern 指定一个从结果集中排除掉的模式。
- Exclude Pattern Flags 排除模式的 Java flags 标准集。
- Include Pattern 指定一个从结果集中要包含的模式。
- Include Pattern Flags 包含模式的 Java flags 标准集。
- JSON Input 一个用来添加 JSON 格式属性的文本框，内容会合并进聚合的定义中，格式如下例：

```
{ "script" : "doc['grade'].value * 1.2" }
```

注意

> Elasticsearch 1.4.3 及以后版本，这个函数需要你开启 dynamic Groovy scripting。

这些参数是否可用，依赖于你选择的聚合函数。

选择 **view options** 更改表格中如下方面：

- Per Page 这个输入框控制表格的翻页。默认值是每页 10 行。

多选框用来控制以下行为：

- Show metrics for every bucket/level 勾选此项用以显示每个 bucket 聚合的中

间结果。

- Show partial rows 勾选此项显示没有数据的行。

注意

> 开启这些行为可能带来性能上的显著影响。

Lines Charts

这个图的 Y 轴是数值维度。该维度有以下聚合可用：

- Count count 聚合返回选中索引模式中元素的原始计数。
- Average 这个聚合返回一个数值字段的 average 。从下拉菜单选择一个字段。
- Sum sum 聚合返回一个数值字段的总和。从下拉菜单选择一个字段。
- Min min 聚合返回一个数值字段的最小值。从下拉菜单选择一个字段。
- Max max 聚合返回一个数值字段的最大值。从下拉菜单选择一个字段。
- Unique Count cardinality 聚合返回一个字段的去重数据值。从下拉菜单选择一个字段。
- Standard Deviation extended stats 聚合返回一个数值字段数据的标准差。从下拉菜单选择一个字段。
- Percentile percentile 聚合返回一个数值字段中值的百分比分布。从下拉菜单选择一个字段，然后在 **Percentiles** 框内指定范围。点击 **X** 移除一个百分比框，点击 **+ Add Percent** 添加一个百分比框。
- Percentile Rank percentile ranks 聚合返回一个数值字段中你指定值的百分位排名。从下拉菜单选择一个字段，然后在 **Values** 框内指定一到多个百分位排名值。点击 **X** 移除一个百分比框，点击 **+Add** 添加一个数值框。

你可以点击 **+ Add Aggregation** 按键添加一个聚合。

buckets 聚合指明从你的数据集中将要检索什么信息。

在你选定一个 buckets 聚合之前，先指定你是要切割单个图的分片，还是切割成多个图形。多图切分必须在其他任何聚合之前要运行。如果你切分图形，你可以选择切分结果是展示成行还是列的形式，点击 **Rows | Columns** 选择器即可。

图形的 X 轴是*buckets* 维度。你可以为 X 轴定义 buckets，同样还可以为图片上的分片区域，或者分割的图片定义 buckets。

该图形的 X 轴支持以下聚合。点击每个聚合的链接查看该聚合的 Elasticsearch 官方文档。

- Date Histogram date histogram 基于数值字段创建，由时间组织起来。你可以指定时间片的间隔，单位包括秒，分，小时，天，星期，月，年。
- Histogram 标准 histogram 基于数值字段创建。为这个字段指定一个整数间隔。勾选 **Show empty buckets** 让直方图中包含空的间隔。

- Range 通过 range 聚合。你可以为一个数值字段指定一系列区间。点击 **Add Range** 添加一堆区间端点。点击红色 **(x)** 符号移除一个区间。
- Date Range date range 聚合计算你指定的时间区间内的值。你可以使用 date math 表达式指定区间。点击 **Add Range** 添加新的区间端点。点击红色 **(/)** 符号移除区间。
- IPv4 Range IPv4 range 聚合用来指定 IPv4 地址的区间。点击 **Add Range** 添加新的区间端点。点击红色 **(/)** 符号移除区间。
- Terms terms 聚合允许你指定展示一个字段的首尾几个元素，排序方式可以是计数或者其他自定义的metric。
- Filters 你可以为数据指定一组 filters。你可以用 query string，也可以用 JSON 格式来指定过滤器，就像在 Discover 页的搜索栏里一样。点击 **Add Filter** 添加下一个过滤器。
- Significant Terms 展示实验性的 significant terms 聚合的结果。

一旦你定义好了一个 X 轴聚合。你可以继续定义子聚合来完善可视化效果。点击 **+ Add Sub Aggregation** 添加子聚合，然后选择 **Split Area** 或者 **Split Chart**，然后从类型菜单中选择一个子聚合。

当一个图形中定义了多个聚合，你可以使用聚合类型右侧的上下箭头来改变聚合的优先级。

你可以点击 **Advanced** 链接显示更多有关聚合的自定义参数：

- Exclude Pattern 指定一个从结果集中排除掉的模式。
- Exclude Pattern Flags 排除模式的 Java flags 标准集。
- Include Pattern 指定一个从结果集中要包含的模式。
- Include Pattern Flags 包含模式的 Java flags 标准集。
- JSON Input 一个用来添加 JSON 格式属性的文本框，内容会合并进聚合的定义中，格式如下例：

```
{ "script" : "doc['grade'].value * 1.2" }
```

注意

> Elasticsearch 1.4.3 及以后版本，这个函数需要你开启 dynamic Groovy scripting。

这些参数是否可用，依赖于你选择的聚合函数。

多选框可以用来控制以下行为：

- Y 轴比例 可以给图形的 Y 轴选择 **linear**, **log** 或 **square root** 三种比例。你可以给指数变化的数据采用 log 函数比例显示，比如复利图表；也可以用平方根(square root)比例显示数值变化差异极大的数据集。这种可变性本身也算变量的一种的数据，又叫异方差数据。比如，身高和体重的数据集，在较矮的区间变化是很小的，但是在较高另一个区间，数据集就是异方差式的。
- Smooth Lines 勾选该项，将图中的数据点用平滑曲线连接。注意：平滑曲线在高峰低谷处给人的印象与实际值有较大偏差。
- Show Connecting Lines 勾选该项，将图中的数据点用折线连接。
- Show Circles 勾选该项，将图中的数据点绘制成一个小圆圈。
- Current time marker 对时序数据，勾选该项可以在当前时刻位置标记一条红线。
- Set Y-Axis Extents 勾选该项，然后在 **y-max** 和 **y-min** 框里输入数值限定 Y 轴为指定数值。
- Show Tooltip 勾选该项显示工具栏。
- Show Legend 勾选该项在图形右侧显示图例。
- Scale Y-Axis to Data Bounds 默认的 Y 轴长度为 0 到数据集的最大值。勾选该项改变 Y 轴的最大和最小值为数据集的返回值。

更新选项后，点击绿色 **Apply changes** 按钮更新你的可视化界面，或者灰色 **Discard changes** 按钮保持原状。

气泡图(Bubble Charts)

通过以下步骤，可以转换折线图成气泡图：

1. 为 Y 轴点击 **Add Metrics** 然后选择 **Dot Size**。
2. 从下拉框里选择一个 metric 聚合函数。
3. 在 **Options** 标签里，去掉 **Show Connecting Lines** 的勾选。
4. 点击 **Apply changes** 按钮。

Markdown 挂件

Markdown 挂件是一个存放 GitHub 风格 Markdown 内容的文本框。Kibana 会渲染你输入的文本，然后在仪表盘上显示渲染结果。你可以点击 **Help** 连接跳转到 help page 查看 GitHub 风格 Markdown 的说明。点击 **Apply** 在预览框查看渲染效果，或者 **Discard** 回退成上一个版本的内容。

Metric

metric 可视化为你选择的聚合显示一个单独的数字：

- Count count 聚合返回选中索引模式中元素的原始计数。
- Average 这个聚合返回一个数值字段的 average 。从下拉菜单选择一个字段。
- Sum sum 聚合返回一个数值字段的总和。从下拉菜单选择一个字段。
- Min min 聚合返回一个数值字段的最小值。从下拉菜单选择一个字段。
- Max max 聚合返回一个数值字段的最大值。从下拉菜单选择一个字段。
- Unique Count cardinality 聚合返回一个字段的去重数据值。从下拉菜单选择一个字段。
- Standard Deviation extended stats 聚合返回一个数值字段数据的标准差。从下拉菜单选择一个字段。
- Percentile percentile 聚合返回一个数值字段中值的百分比分布。从下拉菜单选择一个字段，然后在 **Percentiles** 框内指定范围。点击 **X** 移除一个百分比框，点击 **+ Add Percent** 添加一个百分比框。
- Percentile Rank percentile ranks 聚合返回一个数值字段中你指定值的百分位排名。从下拉菜单选择一个字段，然后在 **Values** 框内指定一到多个百分位排名值。点击 **X** 移除一个百分比框，点击 **+Add** 添加一个数值框。

你可以点击 **+ Add Aggregation** 按键添加一个聚合。你可以点击 **Advanced** 链接显示更多有关聚合的自定义参数：

- Exclude Pattern 指定一个从结果集中排除掉的模式。
- Exclude Pattern Flags 排除模式的 Java flags 标准集。
- Include Pattern 指定一个从结果集中要包含的模式。
- Include Pattern Flags 包含模式的 Java flags 标准集。
- JSON Input 一个用来添加 JSON 格式属性的文本框，内容会合并进聚合的定义中，格式如下例：

```
{ "script" : "doc['grade'].value * 1.2" }
```

注意

> Elasticsearch 1.4.3 及以后版本，这个函数需要你开启 dynamic Groovy scripting。

这些参数是否可用，依赖于你选择的聚合函数。

点击 **view options** 修改显示 metric 的字体大小。

饼图

饼图的分片大小通过 *metrics* 聚合定义。这个维度可以支持以下聚合：

- Count count 聚合返回选中索引模式中元素的原始计数。
- Sum sum 聚合返回一个数值字段的总和。从下拉菜单选择一个字段。
- Unique Count cardinality 聚合返回一个字段的去重数据值。从下拉菜单选择一个字段。

buckets 聚合指明从你的数据集中将要检索什么信息。

在你选定一个 buckets 聚合之前，先指定你是要切割单个图的分片，还是切割成多个图形。多图切分必须在其他任何聚合之前要运行。如果你切分图形，你可以选择切分结果是展示成行还是列的形式，点击 **Rows | Columns** 选择器即可。

你可以为你的饼图指定以下任意 bucket 聚合：

- Date Histogram date histogram 基于数值字段创建，由时间组织起来。你可以指定时间片的间隔，单位包括秒，分，小时，天，星期，月，年。
- Histogram 标准 histogram 基于数值字段创建。为这个字段指定一个整数间隔。勾选 **Show empty buckets** 让直方图中包含空的间隔。
- Range 通过 range 聚合。你可以为一个数值字段指定一系列区间。点击 **Add Range** 添加一堆区间端点。点击红色 **(x)** 符号移除一个区间。
- Date Range date range 聚合计算你指定的时间区间内的值。你可以使用 date math 表达式指定区间。点击 **Add Range** 添加新的区间端点。点击红色 **(/)** 符号移除区间。
- IPv4 Range IPv4 range 聚合用来指定 IPv4 地址的区间。点击 **Add Range** 添加新的区间端点。点击红色 **(/)** 符号移除区间。
- Terms terms 聚合允许你指定展示一个字段的首尾几个元素，排序方式可以是计数或者其他自定义的metric。
- Filters 你可以为数据指定一组 filters。你可以用 query string，也可以用 JSON 格式来指定过滤器，就像在 Discover 页的搜索栏里一样。点击 **Add Filter** 添加下一个过滤器。
- Significant Terms 展示实验性的 significant terms 聚合的结果。

一旦你定义好了一个 X 轴聚合。你可以继续定义子聚合来完善可视化效果。点击 **+ Add Sub Aggregation** 添加子聚合，然后选择 **Split Area** 或者 **Split Chart**，然后从类型菜单中选择一个子聚合。

当一个图形中定义了多个聚合，你可以使用聚合类型右侧的上下箭头来改变聚合的优先级。

你可以点击 **Advanced** 链接显示更多有关聚合的自定义参数：

- Exclude Pattern 指定一个从结果集中排除掉的模式。
- Exclude Pattern Flags 排除模式的 Java flags 标准集。
- Include Pattern 指定一个从结果集中要包含的模式。
- Include Pattern Flags 包含模式的 Java flags 标准集。
- JSON Input 一个用来添加 JSON 格式属性的文本框，内容会合并进聚合的定义中，格式如下例：

```
{ "script" : "doc['grade'].value * 1.2" }
```

注意

> Elasticsearch 1.4.3 及以后版本，这个函数需要你开启 dynamic Groovy scripting。

这些参数是否可用，依赖于你选择的聚合函数。

选择 **view options** 更改表格中如下方面：

- Donut Display the chart as a sliced ring instead of a sliced pie.
- Show Tooltip 勾选该项显示工具栏。
- Show Legend 勾选该项在图形右侧显示图例。

瓦片地图

瓦片地图显示一个由圆圈覆盖着的地理区域。这些圆圈则是由你指定的 buckets 控制的。

瓦片地图的默认 metrics 聚合是 **Count** 聚合。你可以选择下列聚合中的任意一个作为 metrics 聚合：

- Count count 聚合返回选中索引模式中元素的原始计数。
- Average 这个聚合返回一个数值字段的 average 。从下拉菜单选择一个字段。
- Sum sum 聚合返回一个数值字段的总和。从下拉菜单选择一个字段。
- Min min 聚合返回一个数值字段的最小值。从下拉菜单选择一个字段。
- Max max 聚合返回一个数值字段的最大值。从下拉菜单选择一个字段。
- Unique Count cardinality 聚合返回一个字段的去重数据值。从下拉菜单选择一个字段。

buckets 聚合指明从你的数据集中将要检索什么信息。

在你选择 buckets 聚合前，指定你是打算分割图形，还是在单个图形上显示 buckets 为 **Geo Coordinates**。多图切割的聚合必须在最先运行。

瓦片地图使用 **Geohash** 聚合作为他们的初始化聚合。从下拉菜单中选择一个坐标字段。**Precision** 滑动条设置圆圈在地图上显示的颗粒度大小。阅读 geohash grid 聚合的文档，了解每个精度级别的区域细节。Kibana 4.1 目前支持的最大 geohash 长度为 7。

注意

更高的精度意味着同时消耗了浏览器和 ES 集群更多的内存。

一旦你定义好了一个 X 轴聚合。你可以继续定义子聚合来完善可视化效果。点击 **+ Add Sub Aggregation** 添加子聚合，然后选择 **Split Area** 或者 **Split Chart**，然后从类型菜单中选择一个子聚合。

- Date Histogram date histogram 基于数值字段创建，由时间组织起来。你可以指定时间片的间隔，单位包括秒，分，小时，天，星期，月，年。
- Histogram 标准 histogram 基于数值字段创建。为这个字段指定一个整数间

隔。勾选 **Show empty buckets** 让直方图中包含空的间隔。

- Range 通过 range 聚合。你可以为一个数值字段指定一系列区间。点击 **Add Range** 添加一堆区间端点。点击红色 **(x)** 符号移除一个区间。
- Date Range date range 聚合计算你指定的时间区间内的值。你可以使用 date math 表达式指定区间。点击 **Add Range** 添加新的区间端点。点击红色 **(/)** 符号移除区间。
- IPv4 Range IPv4 range 聚合用来指定 IPv4 地址的区间。点击 **Add Range** 添加新的区间端点。点击红色 **(/)** 符号移除区间。
- Terms terms 聚合允许你指定展示一个字段的首尾几个元素，排序方式可以是计数或者其他自定义的metric。
- Filters 你可以为数据指定一组 filters。你可以用 query string，也可以用 JSON 格式来指定过滤器，就像在 Discover 页的搜索栏里一样。点击 **Add Filter** 添加下一个过滤器。
- Significant Terms 展示实验性的 significant terms 聚合的结果。
- Geohash The geohash aggregation displays points based on the geohash coordinates.

你可以点击 **Advanced** 链接显示更多有关聚合的自定义参数：

- Exclude Pattern 指定一个从结果集中排除掉的模式。
- Exclude Pattern Flags 排除模式的 Java flags 标准集。
- Include Pattern 指定一个从结果集中要包含的模式。
- Include Pattern Flags 包含模式的 Java flags 标准集。
- JSON Input 一个用来添加 JSON 格式属性的文本框，内容会合并进聚合的定义中，格式如下例：

```
{ "script" : "doc['grade'].value * 1.2" }
```

注意

> Elasticsearch 1.4.3 及以后版本，这个函数需要你开启 dynamic Groovy scripting。

这些参数是否可用，依赖于你选择的聚合函数。

选择 **Options** 改变表格的如下方面：

Map type

从下拉框选择下面一个选项。

- Shaded Circle Markers 根据 metric 聚合的值显示不同的颜色。
- Scaled Circle Markers 根据 metric 聚合的值显示不同的大小。
- Shaded Geohash Grid 用矩形替换默认的圆形显示 geohash，根据 metric 聚合的值显示不同的颜色。
- Heatmap 热力图可以模糊化圆标而且层叠显示颜色。热力图本身还有如下选项可用：
    - Radius: 设置单个热力图点的大小。
    - Blur: 设置热力图点的模糊度。
    - Maximum zoom: Kibana 的 Tilemap 支持 18 级缩放。该选项设置热力图最大强度下的最高缩放级别。
    - Minimum opacity: 设置数据点的不透明截止位置。
    - Show Tooltip: 勾选该项，让鼠标放在数据点上时显示该点的数据。

Desaturate map tiles

淡化地图颜色，凸显标记的清晰度。

WMS compliant map server

勾选该项，可以配置使用符合 Web Map Service (WMS) 标准的其他第三方地图服务。需要指定一下参数：

- WMS url: WMS 地图服务的 URL；
- WMS layers: 用于可视化的图层列表，逗号分隔。每个地图服务商都会提供自己的图层。
- WMS version: 该服务商采用的 WMS 版本。
- WMS format: 该服务商使用的图片格式。通常来说是 image/png 或 image/jpeg。
- WMS attribution: 可选项。用户自定义字符串，用来显示在图表右下角的属性说明。
- WMS styles: 逗号分隔的风格列表。每个地图服务商都会提供自己的风格选项。

更新选项后，点击绿色 **Apply changes** 按钮更新你的可视化界面，或者灰色 **Discard changes** 按钮保持原状。

Navigating the Map

可视化地图就绪后，你可以通过一下方式探索地图：

- 在地图任意位置点击并按住鼠标后，拖动即可转移地图中心。按住鼠标左键拖拽绘制方框则可以放大选定区域。
- 点击 **Zoom In/Out** 按钮手动修改缩放级别。
- 点击 **Fit Data Bounds** 按钮让地图自动聚焦到至少有一个数据点的地区。
- 点击 **Latitude/Longitude Filter** 按钮，然后在地图上拖拽绘制一个方框，自动生成这个框范围的经纬度过滤器。

竖条图

这个图的 Y 轴是数值维度。该维度有以下聚合可用：

- Count count 聚合返回选中索引模式中元素的原始计数。
- Average 这个聚合返回一个数值字段的 average 。从下拉菜单选择一个字段。
- Sum sum 聚合返回一个数值字段的总和。从下拉菜单选择一个字段。
- Min min 聚合返回一个数值字段的最小值。从下拉菜单选择一个字段。
- Max max 聚合返回一个数值字段的最大值。从下拉菜单选择一个字段。
- Unique Count cardinality 聚合返回一个字段的去重数据值。从下拉菜单选择一个字段。
- Standard Deviation extended stats 聚合返回一个数值字段数据的标准差。从下拉菜单选择一个字段。
- Percentile percentile 聚合返回一个数值字段中值的百分比分布。从下拉菜单选择一个字段，然后在 **Percentiles** 框内指定范围。点击 **X** 移除一个百分比框，点击 **+ Add Percent** 添加一个百分比框。

你可以点击 **+ Add Aggregation** 按键添加一个聚合。

buckets 聚合指明从你的数据集中将要检索什么信息。

在你选定一个 buckets 聚合之前，先指定你是要切割单个图的分片，还是切割成多个图形。多图切分必须在其他任何聚合之前要运行。如果你切分图形，你可以选择切分结果是展示成行还是列的形式，点击 **Rows | Columns** 选择器即可。

图形的 X 轴是*buckets* 维度。你可以为 X 轴定义 buckets，同样还可以为图片上的分片区域，或者分割的图片定义 buckets。

该图形的 X 轴支持以下聚合。点击每个聚合的链接查看该聚合的 Elasticsearch 官方文档。

- Date Histogram date histogram 基于数值字段创建，由时间组织起来。你可以指定时间片的间隔，单位包括秒，分，小时，天，星期，月，年。
- Histogram 标准 histogram 基于数值字段创建。为这个字段指定一个整数间隔。勾选 **Show empty buckets** 让直方图中包含空的间隔。
- Range 通过 range 聚合。你可以为一个数值字段指定一系列区间。点击 **Add Range** 添加一堆区间端点。点击红色 **(x)** 符号移除一个区间。
- Terms terms 聚合允许你指定展示一个字段的首尾几个元素，排序方式可以是

计数或者其他自定义的metric。
- Filters 你可以为数据指定一组 filters。你可以用 query string，也可以用 JSON 格式来指定过滤器，就像在 Discover 页的搜索栏里一样。点击 **Add Filter** 添加下一个过滤器。
- Significant Terms 展示实验性的 significant terms 聚合的结果。

一旦你定义好了一个 X 轴聚合。你可以继续定义子聚合来完善可视化效果。点击 **+ Add Sub Aggregation** 添加子聚合，然后选择 **Split Area** 或者 **Split Chart**，然后从类型菜单中选择一个子聚合。

当一个图形中定义了多个聚合，你可以使用聚合类型右侧的上下箭头来改变聚合的优先级。

你可以点击 **Advanced** 链接显示更多有关聚合的自定义参数：

- Exclude Pattern 指定一个从结果集中排除掉的模式。
- Exclude Pattern Flags 排除模式的 Java flags 标准集。
- Include Pattern 指定一个从结果集中要包含的模式。
- Include Pattern Flags 包含模式的 Java flags 标准集。
- JSON Input 一个用来添加 JSON 格式属性的文本框，内容会合并进聚合的定义中，格式如下例：

```
{ "script" : "doc['grade'].value * 1.2" }
```

注意

> Elasticsearch 1.4.3 及以后版本，这个函数需要你开启 dynamic Groovy scripting。

这些参数是否可用，依赖于你选择的聚合函数。

选择 **view options** 更改表格中如下方面：

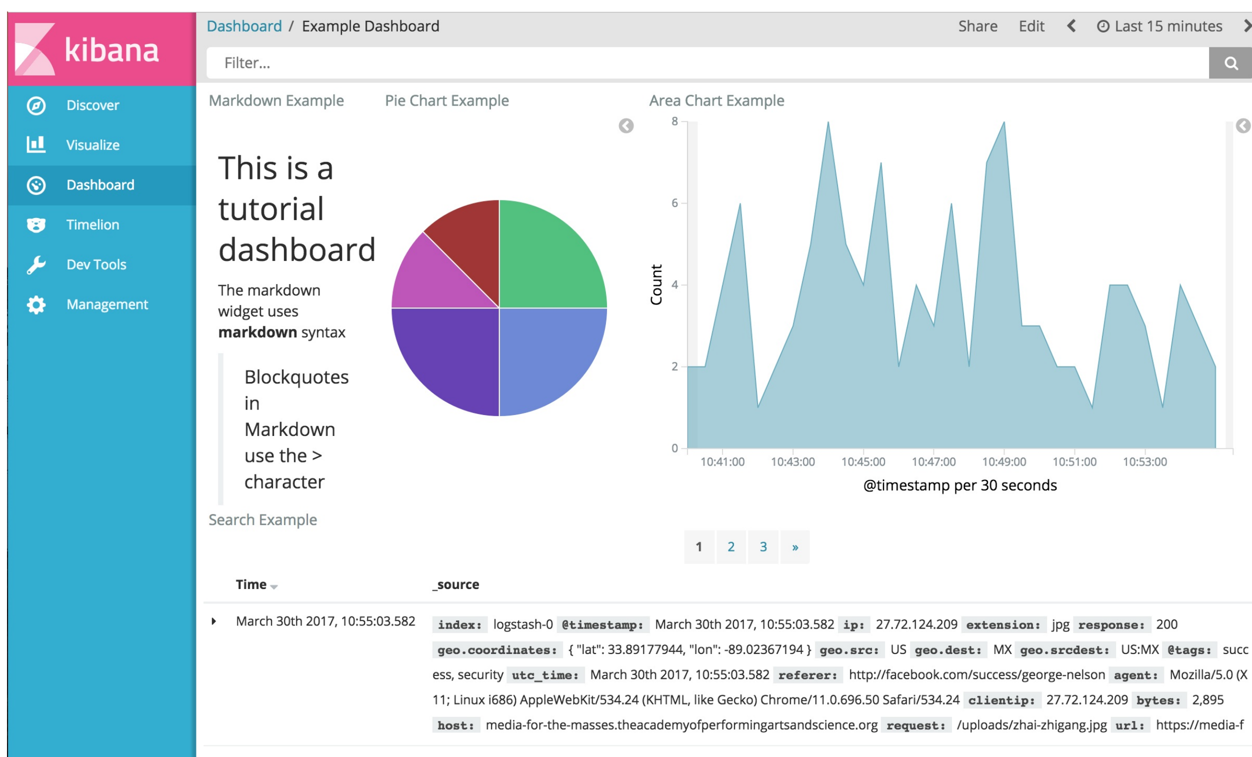
- Bar Mode 当你为自己的图形定义了多个 Y 轴聚合时，你可以用这个选项决定聚合显示的方式：
  - stacked 依次堆叠聚合效果。
  - percentage 每个聚合显示为总和的百分比。
  - grouped 用最低优先级的子聚合的结果做水平分组。

多选框可以用来控制以下行为：

- Show Tooltip 勾选该项显示工具栏。
- Show Legend 勾选该项在图形右侧显示图例。
- Scale Y-Axis to Data Bounds 默认的 Y 轴长度为 0 到数据集的最大值。勾选该项改变 Y 轴的最大和最小值为数据集的返回值。

一个 Kibana *dashboard* 能让你自由排列一组已保存的可视化。然后你可以保存这个仪表板，用来分享或者重载。

简单的仪表板像这样。



开始

要用仪表板，你需要至少有一个已保存的 visualization。

创建一个新的仪表板

你第一次点击 **Dashboard** 标签的时候，Kibana 会显示一个空白的仪表板



通过添加可视化的方式来构建你的仪表板。默认情况下，Kibana 仪表板使用明亮风格。如果你想切换成黑色风格，点击 **Options**，然后勾选 **Use dark theme**。



自动刷新页面

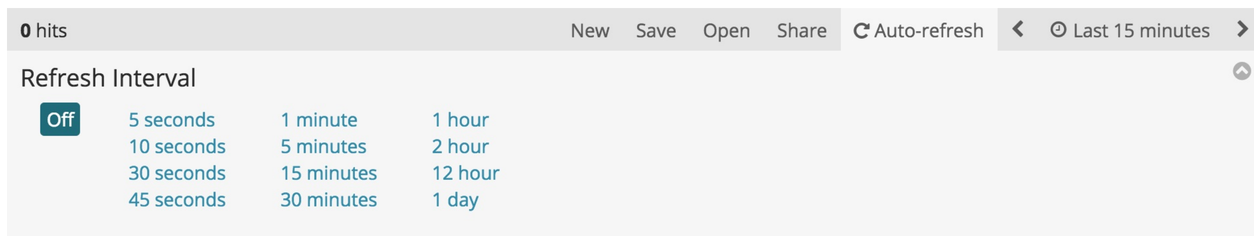
你可以设置页面自动刷新的间隔以查看最新索引进来的数据。这个设置会定期提交搜索请求。

设置刷新间隔后，它会出现在菜单栏的时间过滤器左侧。

要设置刷新间隔：

1. 点击菜单栏右上角的 **Time Filter**
2. 点击 **Refresh Interval** 标签
3. 从列表中选择一个刷新间隔

要自动刷新数据，点击 **Auto-Refresh** 按钮选择自动刷新间隔：



开启自动刷新后，Kibana 顶部菜单栏会显示一个暂停按钮和自动刷新间隔：。点击这个暂停按钮可以暂停自动刷新。

添加可视化到仪表板上

要添加可视化到仪表板上，点击工具栏面板上的 **Add**。从列表中选择一个已保存的可视化。你可以在 **Visualization Filter** 里输入字符串来过滤想要找的可视化。

由你选择的这个可视化会出现在你仪表板上的一个容器*(container)*里。

保存仪表板

要保存仪表板，点击工具栏面板上的 **Save** 按钮，在 **Save As** 栏输入仪表板的名字，然后点击 **Save** 按钮。

加载已保存仪表板

点击 **Open** 按钮显示已存在的仪表板列表。已保存仪表板选择器包括了一个文本栏可以通过仪表板的名字做过滤，还有一个链接到 **Object Editor** 而已管理你的已保存仪表板。你也可以直接点击 **Settings > Edit Saved Objects** 来访问 **Object Editor**。

分享仪表板

你可以分享仪表板给其他用户。可以直接分享 Kibana 的仪表板链接，也可以嵌入到你的网页里。

> 用户必须有 Kibana 的访问权限才能看到嵌入的仪表板。

点击 **Share** 按钮显示 HTML 代码，就可以嵌入仪表板到其他网页里。还带有一个指向仪表板的链接。点击复制按钮 可以复制代码，或者链接到你的黏贴板。

嵌入仪表板

要嵌入仪表板，从 *Share* 页里复制出嵌入代码，然后粘贴进你外部网页应用内即可。

定制仪表板元素

仪表板里的可视化都存在可以调整大小的容器里。接下来会讨论一下容器。

移动容器

点击并按住容器的顶部，就可以拖动容器到仪表板任意位置。其他容器会在必要的时候自动移动，给你在拖动的这个容器空出位置。松开鼠标，容器就会固定在当前停留位置。

改变容器大小

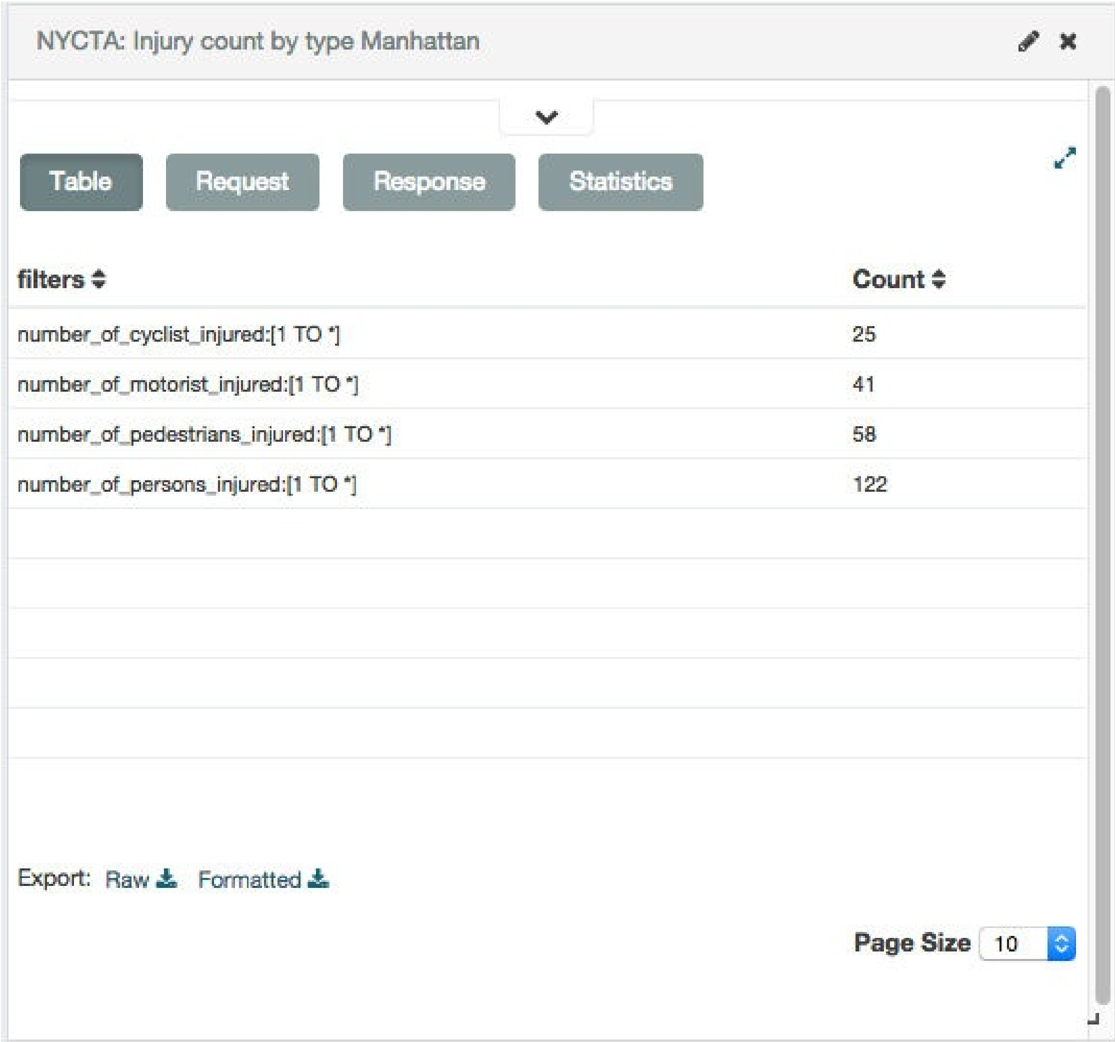
移动光标到容器的右下角，等光标变成指向拐角的方向，点击并按住鼠标，拖动改变容器的大小。松开鼠标，容器就会固定成当前大小。

删除容器

点击容器右上角的 x 图标删除容器。从仪表板删除容器，并不会同时删除掉容器里用到的已存可视化。

查看详细信息

要显示可视化背后的原始数据，点击容器地步的条带。可视化会被有关原始数据详细信息的几个标签替换掉。如下所示：



- 表格(Table)。底层数据的分页展示。你可以通过点击每列顶部的方式给该列数据排序。
- 请求(Request)。发送到服务器的原始请求，以 JSON 格式展示。
- 响应(Response)。从服务器返回的原始响应，以 JSON 格式展示。
- 统计值(Statistics)。和请求响应相关的一些统计值，以数据网格的方式展示。数据报告，请求时间，响应时间，返回的记录条目数，匹配请求的索引模式(index pattern)。

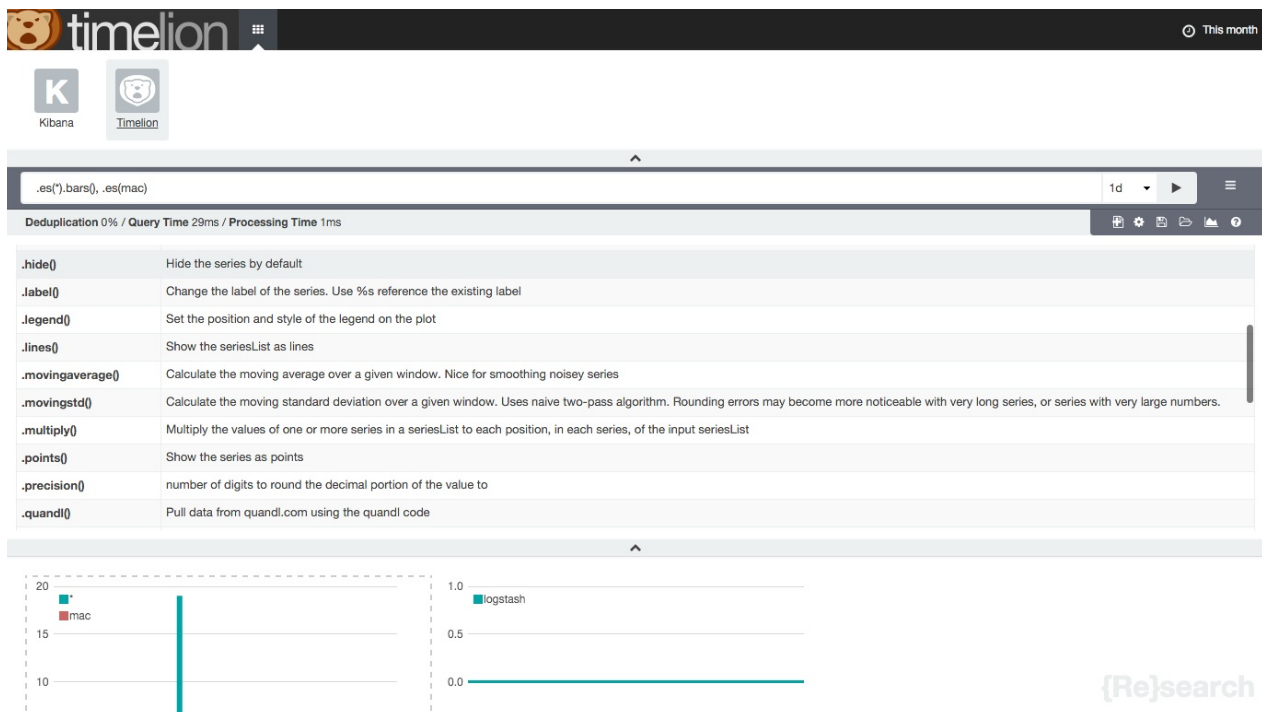
修改可视化

点击容器右上角的 *Edit* 按钮在 Visualize 页打开可视化编辑。

timelion 介绍

Elasticsearch 2.0 开始提供了一个崭新的 pipeline aggregation 特性，但是 Kibana 并没有立刻跟进这方面的意思，相反，Elastic 公司推出了另一个实验室产品：Timelion。最后在 5.0 版中，timelion 成为 Kibana 5 默认分发的一个插件。

timelion 的用法在官博里已经有介绍。尤其是最近两篇如何用 timelion 实现异常告警的文章，更是从 ES 的 pipeline aggregation 细节和场景一路讲到 timelion 具体操作，我这里几乎没有再重新讲一遍 timelion 操作入门的必要了。不过，官方却一直没有列出来 timelion 支持的请求语法的文档，而是在页面上通过点击 Docs 按钮的方式下拉帮助。



timelion 页面设计上，更接近 *Kibana3* 而不是 *Kibana4*。比如 *panel* 分布是通过设置几行几列的数目来固化的；*query* 框是唯一的，要修改哪个 *panel* 的 *query*，鼠标点选一下 *panel*，*query* 就自动切换成这个 *panel* 的了。

为了方便大家在上手之前了解 timelion 能做到什么，今天特意把 timelion 的请求语法所支持的函数分为几类，罗列如下：

可视化效果类

- `.bars($width)`：用柱状图展示数组

- `.lines($width, $fill, $show, $steps)`：用折线图展示数组
- `.points()`：用散点图展示数组
- `.color("#c6c6c6")`：改变颜色
- `.hide()`：隐藏该数组
- `.label("change from %s")`：标签
- `.legend($position, $column)`：图例位置
- `.static(value=1024, label="1k", offset="-1d", fit="scale")`：在图形上绘制一个固定值
- `.value()`：`.static()` 的简写
- `.title(title="qps")`：图表标题
- `.trend(mode="linear", start=0, end=-10)`：采用 linear 或 log 回归算法绘制趋势图
- `.yaxis($yaxis_number, $min, $max, $position)`：设置 Y 轴属性，.yaxis(2) 表示第二根 Y 轴

数据运算类

- `.abs()`：对整个数组元素求绝对值
- `.precision($number)`：浮点数精度
- `.cusum($base)`：数组元素之和，再加上 $base
- `.derivative()`：对数组求导数
- `.divide($divisor)`：数组元素除法
- `.multiply($multiplier)`：数组元素乘法
- `.subtract($term)`：数组元素减法
- `.sum($term)`：数组元素加法
- `.add()`：同 .sum()
- `.plus()`：同 .sum()
- `.first()`：返回第一个元素
- `.movingaverage($window)`：用指定的窗口大小计算移动平均值
- `.mvavg()`：`.movingaverage()` 的简写
- `.movingstd($window)`：用指定的窗口大小计算移动标准差
- `.mvstd()`：`.movingstd()` 的简写
- `.fit($mode)`：使用指定的 fit 函数填充空值。可选项有：average, carry, nearest, none, scale
- `.holt(alpha=0.5, beta=0.5, gamma=0.5, season="1w", sample=2)`：

即 Elasticsearch 的 pipeline aggregation 所支持的 holt-winters 算法

- `.log(base=10)` ：对数
- `.max()` ：最大值
- `.min()` ：最小值
- `.props()` ：附加额外属性，比如 `.props(label=bears!)`
- `.range(max=10, min=1)` ：保持形状的前提下修改最大值最小值
- `.scale_interval(interval="1s")` ：在新间隔下再次统计，比如把一个原本 5min 间隔的 `date_histogram` 改为每秒的结果
- `.trim(start=1, end=-1)` ：裁剪序列值

逻辑运算类

- `.condition(operator="eq", if=100, then=200)` ：支持 eq、ne、lt、gt、lte、gte 等操作符，以及 if、else、then 赋值
- `.if()` ： `.condition()` 的简写

数据源设定类

- `.elasticsearch()` : 从 ES 读取数据
- `.es(q="querystring", metric="cardinality:uid", index="logstash-*", offset="-1d")` : .elasticsearch() 的简写
- `.graphite(metric="path.to.*.data", offset="-1d")` : 从 graphite 读取数据
- `.quandl()` : 从 quandl.com 读取 quandl 码
- `.worldbank_indicators()` : 从 worldbank.org 读取国家数据
- `.wbi()` : `.worldbank_indicators()` 的简写
- `.worldbank()` : 从 worldbank.org 读取数据
- `.wb()` : `.worldbanck()` 的简写

以上所有函数，都在 `kibana/src/core_plugins/timelion/server/series_functions` 目录下实现，每个 js 文件实现一个 `TimelionFunction` 功能。

console 应用

5.0 版本以后，原先 Elasticsearch 的 site plugin 都被废弃掉。其中一些插件作者选择了作为独立的单页应用继续存在，比如前文介绍的 cerebro。而官方插件，则都迁移成为了 Kibana 的 App 扩展。其中，原先归属在 Marvel 中的 Sense 插件，被改名为 Kibana console 应用，在 5.0 版本中默认随 Kibana 分发。

console 应用的操作界面和原先的 sense 几乎一模一样，这里无需赘述：

Dev Tools　　History　Settings　Help

Console

```
# Delete all data in the `website` index
DELETE /website

# Create a document with ID 123
PUT /website/blog/123
{
  "title": "My first blog entry",
  "text":  "Just trying this out...",
  "date":  "2014/01/01"
}

# Search!
GET website/_search
{
  "query": {
    "match": {
      "title": "blog"
    }
  }
}



# Delete all data in the `website` index
DELETE /website

# Create a document with ID 123
PUT /website/blog/123
{
  "title": "My first blog entry",
  "text":  "Just trying this out...",
  "date":  "2014/01/01"
}

# Search!
GET website/_search
{
  "query": {
    "match": {
      "title": "blog"
    }
  }
}
```

要使用 Kibana，你就得告诉它你想要探索的 Elasticsearch 索引是那些，这就要配置一个或者更多的索引模式。此外，你还可以：

- 创建脚本化字段，这个字段可以实时从你的数据中计算出来。你可以浏览这种字段，并且在它基础上做可视化，但是不能搜索这种字段。
- 设置高级选项，比如表格里显示多少行，常用字段显示多少个。修改高级选项的时候要千万小心，因为一个设置很可能跟另一个设置是不兼容的。
- 为生产环境配置 Kibana。

创建一个连接到 **Elasticsearch** 的索引模式

一个索引模式定义了一个或者多个你打算探索的 Elasticsearch 索引。Kibana 会查找匹配指定模式的索引名。模式中的通配符(*)匹配零到多个字符。比如，模式 `myindex-*` 匹配所有名字以 `myindex-` 开头的索引，比如 `myindex-1` 和 `myindex-2` 。

如果你用了事件时间来创建索引名(比如说，如果你是用 Logstash 往 Elasticsearch 里写数据)，索引模式里也可以匹配一个日期格式。在这种情况下，模式的静态文本部分必须用中括号包含起来，日期格式能用的字符，参见表 1 "日期格式码"。

比如， `[logstash-]YYYY.MM.DD` 匹配所有名字以 `logstash-` 为前缀，后面跟上 `YYYY.MM.DD` 格式时间戳的索引，比如 `logstash-2015.01.31` 和 `logstash-2015-02-01` 。

索引模式也可以简单的设置为一个单独的索引名字。

要创建一个连接到 Elasticsearch 的索引模式：

1. 切换到 `Settings > Indices` 标签页。
2. 指定一个能匹配你的 Elasticsearch 索引名的索引模式。默认的，Kibana 会假设你是要处理 Logstash 导入的数据。

> 当你在顶层标签页之间切换的时候，Kibana 会记住你之前停留的位置。比如，如果你在 Settings 标签页查看了一个索引模式，然后切换到 Discover 标签，再切换回 Settings 标签，Kibana 还会显示上次你查看的索引模式。要看到创建模式的表单，需要从索引模式列表里点击 `Add` 按钮。

1. 如果你索引有时间戳字段打算用来做基于事件的对比，勾选 `Index contains time-based events` 然后选择包含了时间戳的索引字段。Kibana 会读取索引

映射，列出所有包含了时间戳的字段供选择。

2. 如果新索引是周期性生成，名字里有时间戳的，勾选 `Use event times to create index names` 和 `Index pattern interval` 选项。这会让 Kibana 只搜索哪些包含了你指定的时间范围内的数据的索引。当你使用 Logstash 往 Elasticsearch 写数据的时候非常有用。
3. 点击 `Create` 添加索引模式。
4. 要设置新模式作为你查看 Discover 页是的默认模式，点击 `favorite` 按钮。

表 1. 日期格式码

格式	描述
M	Month - cardinal: 1 2 3 … 12
Mo	Month - ordinal: 1st 2nd 3rd … 12th
MM	Month - two digit: 01 02 03 … 12
MMM	Month - abbreviation: Jan Feb Mar … Dec
MMMM	Month - full: January February March … December
Q	Quarter: 1 2 3 4
D	Day of Month - cardinal: 1 2 3 … 31
Do	Day of Month - ordinal: 1st 2nd 3rd … 31st
DD	Day of Month - two digit: 01 02 03 … 31
DDD	Day of Year - cardinal: 1 2 3 … 365
DDDo	Day of Year - ordinal: 1st 2nd 3rd … 365th
DDDD	Day of Year - three digit: 001 002 … 364 365
d	Day of Week - cardinal: 0 1 3 … 6
do	Day of Week - ordinal: 0th 1st 2nd … 6th
dd	Day of Week - 2-letter abbreviation: Su Mo Tu … Sa
ddd	Day of Week - 3-letter abbreviation: Sun Mon Tue … Sat
dddd	Day of Week - full: Sunday Monday Tuesday … Saturday
e	Day of Week (locale): 0 1 2 … 6
E	Day of Week (ISO): 1 2 3 … 7
w	Week of Year - cardinal (locale): 1 2 3 … 53
wo	Week of Year - ordinal (locale): 1st 2nd 3rd … 53rd

ww	Week of Year - 2-digit (locale): 01 02 03 … 53
W	Week of Year - cardinal (ISO): 1 2 3 … 53
Wo	Week of Year - ordinal (ISO): 1st 2nd 3rd … 53rd
WW	Week of Year - two-digit (ISO): 01 02 03 … 53
YY	Year - two digit: 70 71 72 … 30
YYYY	Year - four digit: 1970 1971 1972 … 2030
gg	Week Year - two digit (locale): 70 71 72 … 30
gggg	Week Year - four digit (locale): 1970 1971 1972 … 2030
GG	Week Year - two digit (ISO): 70 71 72 … 30
GGGG	Week Year - four digit (ISO): 1970 1971 1972 … 2030
A	AM/PM: AM PM
a	am/pm: am pm
H	Hour: 0 1 2 … 23
HH	Hour - two digit: 00 01 02 … 23
h	Hour - 12-hour clock: 1 2 3 … 12
hh	Hour - 12-hour clock, 2 digit: 01 02 03 … 12
m	Minute: 0 1 2 … 59
mm	Minute - two-digit: 00 01 02 … 59
s	Second: 0 1 2 … 59
ss	Second - two-digit: 00 01 02 … 59
S	Fractional Second - 10ths: 0 1 2 … 9
SS	Fractional Second - 100ths: 0 1 … 98 99
SSS	Fractional Seconds - 1000ths: 0 1 … 998 999
Z	Timezone - zero UTC offset (hh:mm format): -07:00 -06:00 -05:00 .. +07:00
ZZ	Timezone - zero UTC offset (hhmm format): -0700 -0600 -0500 … +0700
X	Unix Timestamp: 1360013296
x	Unix Millisecond Timestamp: 1360013296123

设置默认索引模式

默认索引模式会在你查看 **Discover** 标签的时候自动加载。Kibana 会在 **Settings > Indices** 标签页的索引模式列表里，给默认模式左边显示一个星号。你创建的第一个模式会自动被设置为默认模式。

要设置一个另外的模式为默认索引模式：

1. 进入 `Settings > Indices` 标签页。
2. 在索引模式列表里选择你打算设置为默认值的模式。
3. 点击模式的 `Favorite` 标签。

> 你也可以在 **Advanced > Settings** 里设置默认索引模式。

重加载索引的字段列表

当你添加了一个索引映射，Kibana 自动扫描匹配模式的索引以显示索引字段。你可以重加载索引字段列表，以显示新添加的字段。

重加载索引字段列表，也会重设 Kibana 的常用字段计数器。这个计数器是跟踪你在 Kibana 里常用字段，然后来排序字段列表的。

要重加载索引的字段列表：

1. 进入 `Settings > Indices` 标签页。
2. 在索引模式列表里选择一个索引模式。
3. 点击模式的 `Reload` 按钮。

删除一个索引模式

要删除一个索引模式：

1. 进入 `Settings > Indices` 标签页。
2. 在索引模式列表里选择你打算删除的模式。
3. 点击模式的 `Delete` 按钮。
4. 确认你是想要删除这个索引模式。

创建一个脚本化字段

脚本化字段从你的 Elasticsearch 索引数据中即时计算得来。在 **Discover** 标签页，脚本化字段数据会作为文档数据的一部分显示，而且你还可以在可视化里使用脚本化字段。(脚本化字段的值是在请求的时候计算的，所以它们没有被索引，不能搜索到)

> 即时计算脚本化字段非常消耗资源，会直接影响到 Kibana 的性能。而且记住，Elasticsearch 里没有内置对脚本化字段的验证功能。如果你的脚本有bug，你会在查看动态生成的数据时看到 exception。

脚本化字段使用 Lucene 表达式语法。更多细节，请阅读 Lucene Expressions Scripts。

你可以在表达式里引用任意单个数值类型字段，比如：

```
doc['field_name'].value
```

要创建一个脚本化字段：

1. 进入 `Settings > Indices`
2. 选择你打算添加脚本化字段的索引模式。
3. 进入模式的 `Scripted Fields` 标签。
4. 点击 `Add Scripted Field` 。
5. 输入脚本化字段的名字。
6. 输入用来即时计算数据的表达式。
7. 点击 `Save Scripted Field` .

有关 Elasticsearch 的脚本化字段的更多细节，阅读 Scripting。

更新一个脚本化字段

要更新一个脚本化字段：

1. 进入 `Settings > Indices` 。
2. 点击你要更新的脚本化字段的 `Edit` 按钮。
3. 完成变更后点击 `Save Scripted Field` 升级。

> 注意 Elasticsearch 里没有内置对脚本化字段的验证功能。如果你的脚本有bug，你会在查看动态生成的数据时看到 exception。

删除一个脚本化字段

要删除一个脚本化字段：

1. 进入 `Settings > Indices` 。
2. 点击你要删除的脚本化字段的 `Delete` 按钮。
3. 确认你确实想删除它。

设置高级参数

高级参数页允许你直接编辑那些控制着 Kibana 应用行为的设置。比如，你可以修改显示日期的格式，修改默认的索引模式，设置十进制数值的显示精度。

> 修改高级参数可能带来意想不到的后果。如果你不确定自己在做什么，最好离开这个设置页面。

要设置高级参数：

1. 进入 `Settings > Advanced` 。
2. 点击你要修改的选项的 `Edit` 按钮。
3. 给这个选项输入一个新的值。
4. 点击 `Save` 按钮。

管理已保存的搜索，可视化和仪表板

你可以从 **Settings > Objects** 查看，编辑，和删除已保存的搜索，可视化和仪表板。

查看一个已保存的对象会显示在 **Discover**, **Visualize** 或 **Dashboard** 页里已选择的项目。要查看一个已保存对象：

1. 进入 `Settings > Objects` 。
2. 选择你想查看的对象。
3. 点击 `View` 按钮。

编辑一个已保存对象让你可以直接修改对象定义。你可以修改对象的名字，添加一段说明，以及修改定义这个对象的属性的 JSON。

如果你尝试访问一个对象，而它关联的索引已经被删除了，Kibana 会显示这个对象的编辑(Edit Object)页。你可以：

- 重建索引这样就可以继续用这个对象。
- 删除对象，然后用另一个索引重建对象。
- 在对象的 `kibanaSavedObjectMeta.searchSourceJSON` 里修改引用的索引名，指向一个还存在的索引模式。这个在你的索引被重命名了的情况下非常有用。

> 对象属性没有验证机制。提交一个无效的变更会导致对象不可用。通常来说，你还是应该用 Discover, Visualize 或 Dashboard 页面来创建新对象而不是直接编辑已存在的对象。

要编辑一个已保存的对象：

1. 进入 `Settings > Objects` 。
2. 选择你想编辑的对象。
3. 点击 `Edit` 按钮。
4. 修改对象定义。
5. 点击 `Save Object` 按钮。

要删除一个已保存的对象：

1. 进入 `Settings > Objects` 。
2. 选择你想删除的对象。
3. 点击 `Delete` 按钮。
4. 确认你确实想删除这个对象。

设置 kibana 服务器属性

Kibana 服务器在启动的时候会从 `kibana.yml` 文件读取属性设置。默认设置是运行在 `localhost:5601` 。要变更主机或端口，或者连接远端主机上的 Elasticsearch，你都需要更新你的 `kibana.yml` 文件。你还可以开启 SSL 或者设置其他一系列选项：

表 2. Kibana 服务器属性

属性	描述
server.port	Kibana 服务器运行的端口。默认： `5601` 。

server.host	Kibana 服务器监听的地址。默认： `"0.0.0.0"`
server.defaultRoute	进入 Kibana 时默认跳转的地址。默认为 `/app/kibana` 。
server.ssl.enabled	是否开启 Kibana 服务器的 SSL 验证。
server.ssl.key	Kibana 服务器的密钥文件路径。设置用来加密浏 Kibana 之间的通信。默认：none。
server.ssl.certificate	Kibana 服务器的证书文件路径。设置用来加密浏 Kibana 之间的通信。默认：none。
pid.file	你想用来存进程 ID 文件的位置。如果没有指定， 存在 `/var/run/kibana.pid` 。
kibana.index	保存搜索，可视化，仪表板信息的索引的名字。 认： `.kibana` 。
kibana.defaultAppId	进入 Kibana App 后默认显示的页面。默 认： `discover` 。
tilemap.url	用来显示瓦片地图的服务接口 URL。想使用高德 者可以设置 为： `"http://webrd02.is.autonavi.com/app` `lang=zh_cn&size=1&scale=1&style=7&x={x}&` `{y}&z={z}"` 。
elasticsearch.url	你想请求的索引存在哪个 Elasticsearch 实例上。 认： `"http://localhost:9200"` 。
elasticsearch.preserveHost	默认的，浏览器请求中的主机名即作为 Kibana 发 Elasticsearch 时请求的主机名。如果你设置这个 `false` , Kibana 会改用 `elasticsearch.url` 名。你应该不用担心这个设置 —— 直接用默认即 认： `true` 。
elasticsearch.requestTimeout	等待 Kibana 后端或 Elasticsearch 的响应的超时 位毫秒。默认： `30000` 。
elasticsearch.shardTimeout	Elasticsearch 等待分片响应的超时时间。设置为 闭超时控制。默认： `0` 。
elasticsearch.ssl.key	用来加密 Kibana 和 Elasticsearch 之间的通信的 件。默认：none。
elasticsearch.ssl.certificate	用来加密 Kibana 和 Elasticsearch 之间的通信的 件。默认：none。
elasticsearch.tribe.url	如果使用 Elasticsearch 的 Tribe Node 查询多个 据，需配置 Tribe Node 的地址。

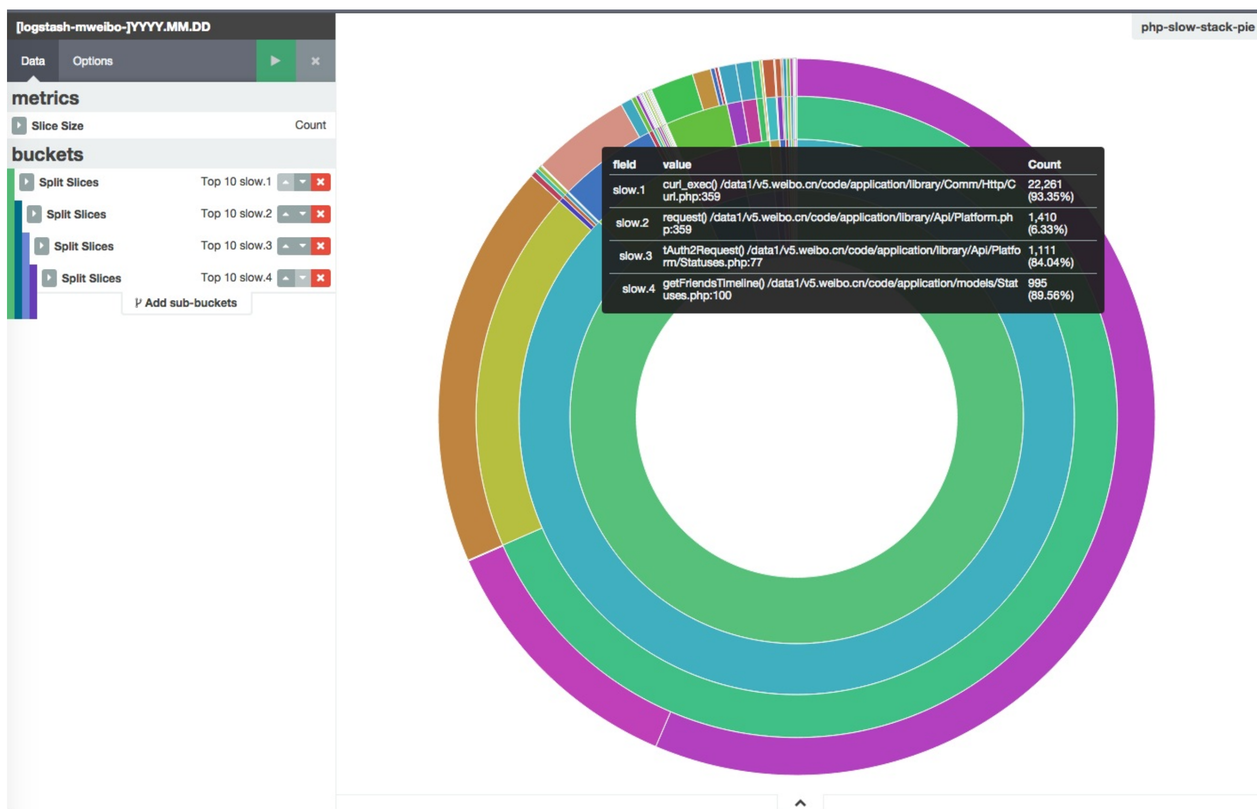
常见 sub aggs 示例

本章开始，就提到 K4/5 和 K3 的区别，在 K4/5 中，即便介绍完了全部 visualize 的配置项，也不代表用户能立刻上手配置出来和 K3 一样的面板。所以本节，会以几个具体的日志分析需求为例，演示在 K4 中，利用 Elasticsearch 1.0 以后提供的 Aggregation 特性，能够做到哪些有用的可视化效果。

函数调用链堆栈

本书之前已经介绍过 logstash 如何利用 multiline 或者 log4j 插件解析函数堆栈。那么，对函数堆栈，我们除了对底层函数做基础的 topN 排序，还能深入发掘出来什么信息呢？

下图是一个 PHP 慢函数堆栈的可视化统计：



该图利用了 Kibana4 的 sub aggs 特性。按照分层次的函数堆栈，逐层做 terms agg。得到一个类似火焰图效果的千层饼效果。

和火焰图不同的是，千层饼并不能自动深入到函数堆栈的全部层次，需要自己手动指定聚合到第几层。考虑到重复操作在页面上不是很方便。可以利用 Kibana4 的 url 特性，直接修改地址生成效果。上图的 url 如下：

```
http://k4domain:5601/#/visualize/edit/php-slow-stack-pie?_g=()&_a=(filters:!(),linked:!t,query:(query_string:(query:'*')),vis:(aggs:!((id:'1',params:(),schema:metric,type:count),(id:'2',params:(field:slow.1,order:desc,orderBy:'1',size:10),schema:segment,type:terms),(id:'3',params:(field:slow.2,order:desc,orderBy:'1',size:10),schema:segment,type:terms),(id:'4',params:(field:slow.3,order:desc,orderBy:'1',size:10),schema:segment,type:terms),(id:'5',params:(field:slow.4,order:desc,orderBy:'1',size:10),schema:segment,type:terms)),listeners:(),params:(addLegend:!f,addTooltip:!t,defaultYExtents:!f,isDonut:!t,shareYAxis:!t,spyPerPage:10),type:pie))
```

可以看到，如果打算增减堆栈的聚合层次，对应增减一段 `(id:'5',params:(field:slow.4,order:desc,orderBy:'1',size:10),` 就可以了。

作为固定可视化分析模式的另一种分享办法，还可以导出该 visualize object 在 `.kibana` 索引中的 JSON 记录。这样其他人只需要原样再导入到自己的 `.kibana` 索引即可：

```
# curl 127.0.0.1:9200/.kibana/visualization/php-slow-stack-pie/_source
{"title":"php-slow-stack-pie","visState":"{\"aggs\":[{\"id\":\"1\",\"params\":{},\"schema\":\"metric\",\"type\":\"count\"},{\"id\":\"2\",\"params\":{\"field\":\"slow.1\",\"order\":\"desc\",\"orderBy\":\"1\",\"size\":10},\"schema\":\"segment\",\"type\":\"terms\"},{\"id\":\"3\",\"params\":{\"field\":\"slow.2\",\"order\":\"desc\",\"orderBy\":\"1\",\"size\":10},\"schema\":\"segment\",\"type\":\"terms\"},{\"id\":\"4\",\"params\":{\"field\":\"slow.3\",\"order\":\"desc\",\"orderBy\":\"1\",\"size\":10},\"schema\":\"segment\",\"type\":\"terms\"},{\"id\":\"5\",\"params\":{\"field\":\"slow.4\",\"order\":\"desc\",\"orderBy\":\"1\",\"size\":10},\"schema\":\"segment\",\"type\":\"terms\"}],\"listeners\":{},\"params\":{\"addLegend\":false,\"addTooltip\":true,\"defaultYExtents\":false,\"isDonut\":true,\"shareYAxis\":true,\"spyPerPage\":10},\"type\":\"pie\"}","description":"","savedSearchId":"php-fpm-slowlog","version":1,"kibanaSavedObjectMeta":{"searchSourceJSON":"{\"filter\":[]}"}}
```

上面记录中可以看到，这个 visualize 还关联了一个 savedSearch，那么同样，再从 `.kibana` 索引里把这个内容也导出：

```
# curl 127.0.0.1:9200/.kibana/search/php-fpm-slowlog/_source
{"title":"php-fpm-slowlog","description":"","hits":0,"columns":[
"_source"],"sort":["@timestamp","desc"],"version":1,"kibanaSaved
ObjectMeta":{"searchSourceJSON":"{\n  \"index\": \"[logstash-mwe
ibo-]YYYY.MM.DD\",\n  \"highlight\": {\n    \"pre_tags\": [\n
   \"@kibana-highlighted-field@\"\n    ],\n    \"post_tags\": [\
n      \"@/kibana-highlighted-field@\"\n    ],\n    \"fields\":
{\n      \"*\": {}\n    }\n  },\n  \"filter\": [\n    {\n      \
"meta\": {\n        \"index\": \"[logstash-mweibo-]YYYY.MM.DD\",
\n        \"negate\": false,\n        \"key\": \"_type\",\n
   \"value\": \"php-fpm-slow\",\n        \"disabled\": false\n
    },\n      \"query\": {\n        \"match\": {\n          \"_t
ype\": {\n            \"query\": \"php-fpm-slow\",\n
\"type\": \"phrase\"\n          }\n        }\n      }\n    }\n
],\n  \"query\": {\n    \"query_string\": {\n      \"query\": \"
*\",\n      \"analyze_wildcard\": true\n    }\n  }\n}"}}
```

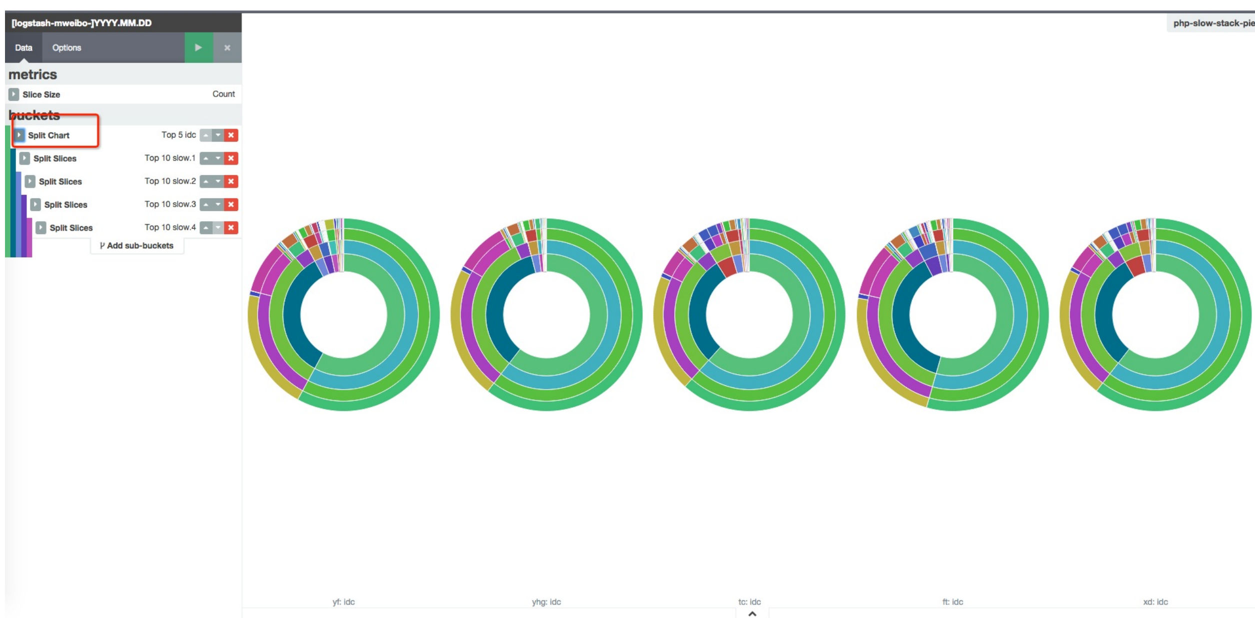
这个内容看起来有点怪怪的，其实把 `searchSourceJSON` 字符串复制出来，在终端下贴到 `echo -ne` 命令后面，回车即可看到其实是这样：

```
{
  "index": "[logstash-mweibo-]YYYY.MM.DD",
  "highlight": {
    "pre_tags": [
      "@kibana-highlighted-field@"
    ],
    "post_tags": [
      "@/kibana-highlighted-field@"
    ],
    "fields": {
      "*": {}
    }
  },
  "filter": [
    {
      "meta": {
        "index": "[logstash-mweibo-]YYYY.MM.DD",
        "negate": false,
        "key": "_type",
        "value": "php-fpm-slow",
        "disabled": false
      },
      "query": {
        "match": {
          "_type": {
            "query": "php-fpm-slow",
            "type": "phrase"
          }
        }
      }
    }
  ],
  "query": {
    "query_string": {
      "query": "*",
      "analyze_wildcard": true
    }
  }
}
```

分图统计

上一节我们展示了 sub aggs 在饼图上的效果。不过这多层 agg，其实用的是同一类数据。如果在聚合中，要加上一些完全不同纬度的数据，还是在单一的图片上继续累加就不是很直观了。比如说，还是上一节用到的 PHP 慢函数堆栈。我们可以根据机房做一下拆分。由于代码部署等主动变更都是有灰度部署的，一旦发现某机房有异常，就可以及时处理了。

同样还是新建 sub aggs，但是在开始，选择 split chart 而不是 split slice。



从 URL 里可以看到，分图的 aggs 是 `schema:split`，而饼图分片的 aggs 是 `schema:segment`：

```
http://k4domain:5601/#/visualize/edit/php-slow-stack-pie?_g=(ref
reshInterval:(display:Off,pause:!f,section:0,value:0),time:(from
:now-12h,mode:quick,to:now))&_a=(filters:!(),linked:!t,query:(qu
ery_string:(query:'*')),vis:(aggs:!((id:'1',params:(),schema:met
ric,type:count),(id:'6',params:(field:idc,order:desc,orderBy:'1'
,row:!f,size:5),schema:split,type:terms),(id:'2',params:(field:s
low.1,order:desc,orderBy:'1',size:10),schema:segment,type:terms)
,(id:'3',params:(field:slow.2,order:desc,orderBy:'1',size:10),sc
hema:segment,type:terms),(id:'4',params:(field:slow.3,order:desc
,orderBy:'1',size:10),schema:segment,type:terms),(id:'5',params:
(field:slow.4,order:desc,orderBy:'1',size:10),schema:segment,typ
e:terms)),listeners:(),params:(addLegend:!f,addTooltip:!t,defaul
tYExtents:!f,isDonut:!t,shareYAxis:!t,spyPerPage:10),type:pie))
```

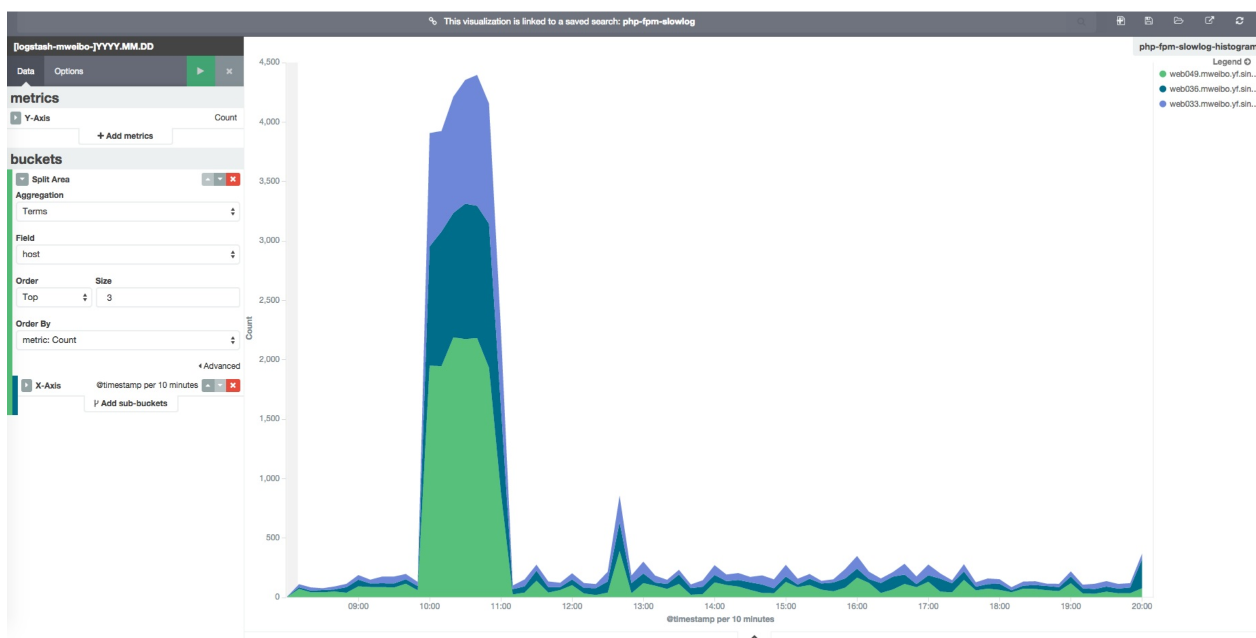
TopN 的时序趋势图

TopN 的时序趋势图是将 Elastic Stack 用于监控场景最常用的手段。乃至在 Kibana3 时代，开发者都通过在 Query 框上额外定义 TopN 输入的方式提供了这个特性。不过在 Kibana4 中，因为 sub aggs 的依次分桶原理，TopN 时序趋势图又有了新的特点。

Kibana3 中，请求实质是先单独发起一次 termFacet 请求得到 topN，然后再发起带有 facetFilter 的 dateHistogramFacet ，分别请求每个 term 的时序。那么同样的原理，迁移到 Kibana4 中，就是先利用一次 termAgg 分桶后，再每个桶内做 dateHistogramAgg 了。对应的 Kibana4 地址为：

```
http://k4domain:5601/#/visualize/edit/php-fpm-slowlog-histogram?
_g=(refreshInterval:(display:Off,pause:!f,section:0,value:0),tim
e:(from:now-12h,mode:quick,to:now))&_a=(filters:!(),linked:!t,qu
ery:(query_string:(query:'*')),vis:(aggs:!((id:'1',params:(),sch
ema:metric,type:count),(id:'3',params:(field:host,order:desc,ord
erBy:'1',size:3),schema:group,type:terms),(id:'2',params:(custom
Interval:'2h',extended_bounds:(),field:'@timestamp',interval:aut
o,min_doc_count:1),schema:segment,type:date_histogram)),listener
s:(),params:(addLegend:!t,addTimeMarker:!f,addTooltip:!t,default
YExtents:!t,interpolate:linear,mode:stacked,scale:linear,setYExt
ents:!f,shareYAxis:!t,smoothLines:!f,times:!(),yAxis:()),type:ar
ea))
```

效果如下：

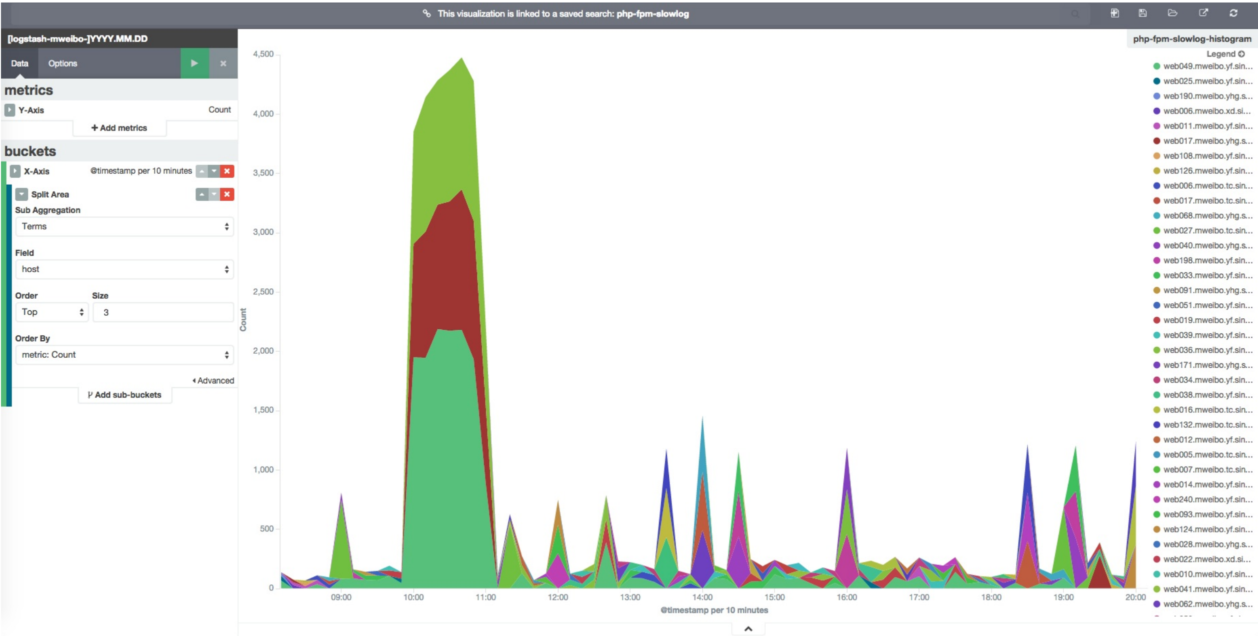


可以看到图上就是 3 根线，分别代表 top3 的 host 的时序。

一般来说，这样都是够用的。不过如果经常有 host 变动的时候，在这么大的一个时间范围内，求一次总的 topN，可能就淹没了一些瞬间的变动了。所以，在 Kibana4 上，我们可以把 sub aggs 的顺序颠倒一下。先按 dateHistogramAgg 分桶，再在每个时间桶内，做 termAgg。对应的 Kibana4 地址为：

```
http://k4domain:5601/#/visualize/edit/php-fpm-slowlog-histogram?_g=(refreshInterval:(display:Off,pause:!f,section:0,value:0),time:(from:now-12h,mode:quick,to:now))&_a=(filters:!(),linked:!t,query:(query_string:(query:'*')),vis:(aggs:!((id:'1',params:(),schema:metric,type:count),(id:'2',params:(customInterval:'2h',extended_bounds:(),field:'@timestamp',interval:auto,min_doc_count:1),schema:segment,type:date_histogram),(id:'3',params:(field:host,order:desc,orderBy:'1',size:3),schema:group,type:terms)),listeners:(),params:(addLegend:!t,addTimeMarker:!f,addTooltip:!t,defaultYExtents:!t,interpolate:linear,mode:stacked,scale:linear,setYExtents:!f,shareYAxis:!t,smoothLines:!f,times:!(),yAxis:()),type:area))
```

可以对比一下前一条 url，其中就是把 id 为 2 和 3 的两段做了对调。而最终效果如下：


This visualization is linked to a saved search: php-fpm-slowlog
[logstash-mweibo-]YYYY.MM.DD
Data
Options
metrics
Y-Axis
Count
+ Add metrics
buckets
X-Axis
@timestamp per 10 minutes
Split Area
Sub Aggregation
Terms
Field
host
Order
Size
Top
3
Order By
metric: Count
Advanced
Add sub-buckets
4,500
4,000
3,500
3,000
2,500
2,000
1,500
1,000
500
0
Count
09:00
10:00
11:00
12:00
13:00
14:00
15:00
16:00
17:00
18:00
19:00
20:00
@timestamp per 10 minutes
php-fpm-slowlog-histogram
Legend
web049.mweibo.yf.sin...
web025.mweibo.yf.sin...
web190.mweibo.yhg.s...
web006.mweibo.xd.si...
web011.mweibo.yf.sin...
web017.mweibo.yhg.s...
web108.mweibo.yf.sin...
web126.mweibo.yf.sin...
web006.mweibo.tc.sin...
web017.mweibo.tc.sin...
web068.mweibo.yhg.s...
web027.mweibo.tc.sin...
web040.mweibo.yhg.s...
web198.mweibo.yf.sin...
web033.mweibo.yf.sin...
web091.mweibo.yhg.s...
web051.mweibo.yf.sin...
web019.mweibo.yf.sin...
web039.mweibo.yf.sin...
web036.mweibo.yf.sin...
web171.mweibo.yhg.s...
web034.mweibo.yf.sin...
web038.mweibo.yf.sin...
web016.mweibo.tc.sin...
web132.mweibo.tc.sin...
web012.mweibo.yf.sin...
web005.mweibo.tc.sin...
web007.mweibo.tc.sin...
web014.mweibo.yf.sin...
web240.mweibo.yf.sin...
web093.mweibo.yf.sin...
web124.mweibo.yf.sin...
web028.mweibo.yhg.s...
web022.mweibo.xd.si...
web010.mweibo.yf.sin...
web041.mweibo.yf.sin...
web062.mweibo.yhg.s...

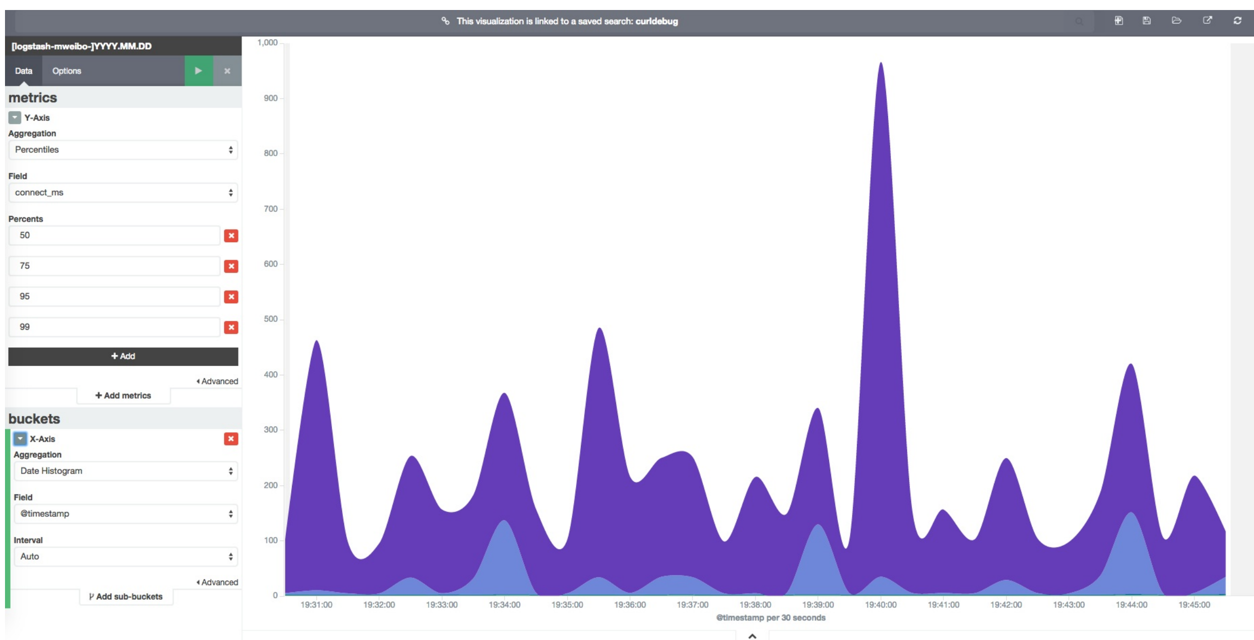
差距多么明显！

响应时间的百分占比趋势图

时序图除了上节展示的最基本的计数以外，还可以在 Y 轴上使用其他数值统计结果。最常见的，比如访问日志的平均响应时间。但是平均值在数学统计中，是一个非常不可信的数据。稍微几个远离置信区间的数值就可以严重影响到平均值。所以，在评价数值的总体分布情况时，更推荐采用四分位数。也就是 25%，50%，75%。在可视化方面，一般采用箱体图方式。

Kibana4 没有箱体图的可视化方式。不过采用线图，我们一样可以做到类似的效果。

在上一章的时序数据基础上，改变 Y 轴数据源，选择 Percentile 方式，然后输入具体的四分位。运行渲染即可。



对比新的 URL，可以发现变化的就是 id 为 1 的片段。变成了 `(id:'1',params:(field:connect_ms,percents:!(50,75,95,99)),schema:metric,type:percentiles)` ：

```
http://k4domain:5601/#/visualize/create?type=area&savedSearchId=
curldebug&_g=()&_a=(filters:!(),linked:!t,query:(query_string:(q
uery:'*')),vis:(aggs:!((id:'1',params:(field:connect_ms,percents
:!(50,75,95,99)),schema:metric,type:percentiles),(id:'2',params:
(customInterval:'2h',extended_bounds:(),field:'@timestamp',inter
val:auto,min_doc_count:1),schema:segment,type:date_histogram)),l
isteners:(),params:(addLegend:!f,addTimeMarker:!f,addTooltip:!t,
defaultYExtents:!f,interpolate:linear,mode:stacked,scale:linear,
setYExtents:!f,shareYAxis:!t,smoothLines:!t,times:!(),yAxis:()),
type:area))
```

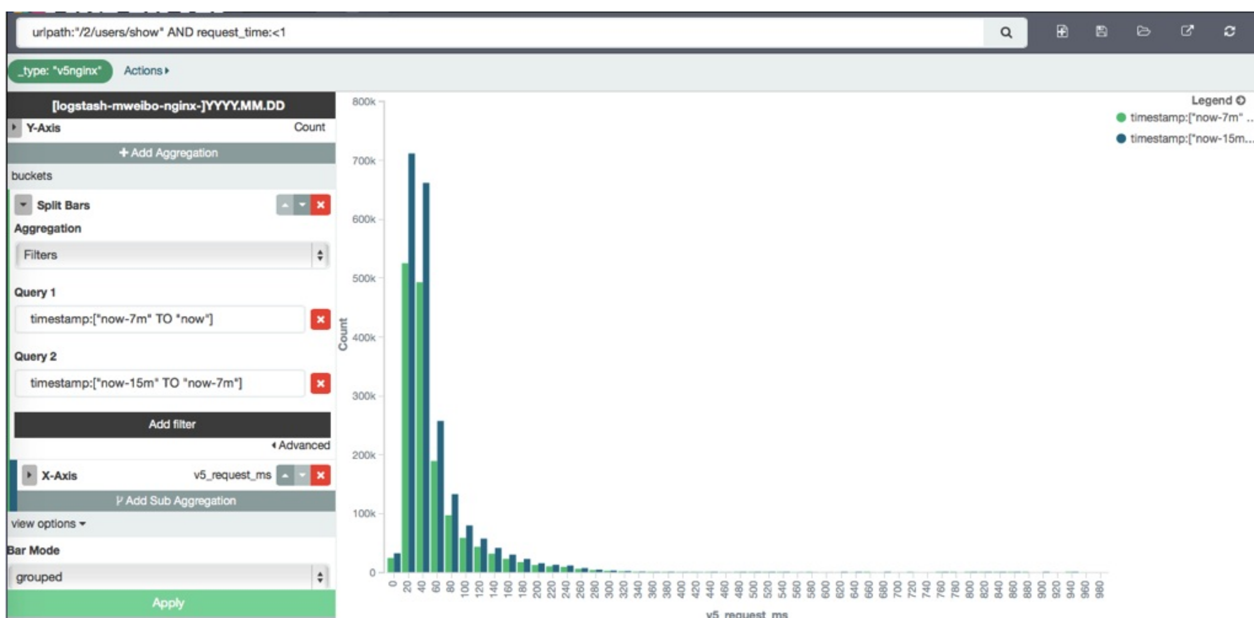
实践表明，在访问日志数据上，平均数一般相近于 75% 的四分位数。所以，为了更细化性能情况，我们可以改用诸如 90%，95% 这样的百分位。

此外，从 Kibana4.1 开始，新加入了 *Percentile_rank* 聚合支持。可以在 Y 轴数据源里选择这种聚合，输入具体的响应时间，比如 2s。则可视化数据变成 2s 内完成的响应数占总数的百分比的趋势图。

响应时间的概率分布在不同时段的相似度对比

前面已经用百分位的时序，展示如何更准确的监控时序数据的波动。那么，还能不能更进一步呢？在制定 SLA 的时候，制定报警阈值的时候，怎么才能确定当前服务的拐点？除了压测以外，我们还可以拿线上服务的实际数据计算一下概率分布。Kibana4 对此提供了直接的支持，我们可以以数值而非时间作为 X 轴数据。

那么进一步，我们怎么区分不同产品在同一时间，或者相同产品在不同时间，性能上有无渐变到质变的可能？这里，我们可以采用 `grouped` 方式，来排列 filter aggs 的结果：



我们可以看出来，虽然两个时间段，响应时间是有一定差距的，但是是整体性的抬升，没有明显的异变。

当然，如果觉得目测不靠谱的，可以把两组数值拿下来，通过 PDL、scipy、matlab、R 等工具做具体的差异显著性检测。这就属于后续的二次开发了。

filter 中，可以写任意的 query string 语法。不限于本例中的时间段：

```
http://k4domain:5601/#/visualize/create?type=histogram&indexPatt
ern=%5Blogstash-mweibo-nginx-%5DYYYY.MM.DD&_g=()&_a=(filters:!()
,linked:!f,query:(query_string:(analyze_wildcard:!t,query:'reque
st_time:%3C50')),vis:(aggs:!((id:'1',params:(),schema:metric,typ
e:count),(id:'3',params:(filters:!((input:(query:(query_string:(
analyze_wildcard:!t,query:'@timestamp:%5B%22now-7m%22%20TO%20%22
now%22%5D')))),(input:(query:(query_string:(analyze_wildcard:!t,
query:'@timestamp:%5B%22now-15m%22%20TO%20%22now-7m%22%7D'))))))
,schema:group,type:filters),(id:'2',params:(extended_bounds:(),f
ield:request_time,interval:1),schema:segment,type:histogram)),li
steners:(),params:(addLegend:!t,addTimeMarker:!f,addTooltip:!t,d
efaultYExtents:!f,mode:grouped,scale:linear,setYExtents:!f,share
YAxis:!t,spyPerPage:10,times:!(),yAxis:()),type:histogram))
```

源码剖析

Kibana 4 开始采用 angular.js + node.js 框架编写。其中 node.js 主要提供两部分功能，给 Elasticsearch 做搜索请求转发代理，以及 auth、ssl、setting 等操作的服务器后端。5.0 版本在这些基础构成方面没有太大变化。

本章节假设你已经对 angular 有一定程度了解。所以不会再解释其中 angular 的 route，controller，directive，service，factory 等概念。

如果打算迁移 kibana 3 的 CAS 验证功能到 Kibana 4 或 5 版本，那么可以稍微了解一下 Hapi.JS 框架的认证扩展，相信可以很快修改成功。

本章主要还是集中在前端 kibana 页面功能的实现上。

参考阅读

- 在 Elastic{ON} 大会上，也有专门针对 Kibana 4 源码和二次开发入门的演讲。请参阅：https://speakerdeck.com/elastic/the-contributors-guide-to-the-kibana-galaxy
- 专业的前端工程师怎么看Kibana 4 的代码的：http://www.debuggerstepthrough.com/2015/04/reviewing-kibana-4s-client-side-code.html。
- 如何通过 Hapi 的 Proxy 功能实现认证：kbn-authentication-plugin

kibana_index 结构

包括有以下 type：

config

_id 为 kibana5 的 version。内容主要是 defaultIndex，设置默认的 index_pattern.

search

_id 为 discover 上保存的搜索名称。内容主要是 title，column，sort，version，description，hits 和 kibanaSavedObjectMeta。kibanaSavedObjectMeta 内是一个 searchSourceJSON，保存搜索 json 的字符串。

visualization

_id 为 visualize 上保存的可视化名称。内容包括 title，savedSearchId，description，version，kibanaSavedObjectMeta 和 visState，uiStateJSON。其中 visState 里保存了 聚合 json 的字符串。如果绑定了已保存的搜索，那么把其在 search 类型里的 _id 存在 savedSearchId 字段里，如果是从新搜索开始的，那么把搜索 json 的字符串直接存在自己的 kibanaSavedObjectMeta 的 searchSourceJSON 里。

dashboard

_id 为 dashboard 上保存的仪表盘名称。内容包括 title, version，timeFrom，timeTo，timeRestore，uiStateJSON，optionsJSON，hits，refreshInterval，panelsJSON 和 kibanaSavedObjectMeta。其中 panelsJSON 是一个数组，每个元素是一个 panel 的属性定义。定义包括有：

- type: 具体加载的 app 类型，就默认来说，肯定就是 search 或者 visualization 之一。
- id: 具体加载的 app 的保存 id。也就是上面说过的，它们在各自类型下的

`_id` 内容。
- size_x: panel 的 X 轴长度。Kibana 采用 gridster 库做挂件的动态划分，默认为 3。
- size_y: panel 的 Y 轴长度。默认为 2。
- col: panel 的左边侧起始位置。Kibana 指定 col 最大为 12。每行第一个 panel 的 col 就是 1，假如它的 size_x 是 4，那么第二个 panel 的 col 就是 5。
- row: panel 位于第几行。gridster 默认的 row 最大为 15。

index-pattern

_id 为 setting 中设置的 index pattern。内容主要是匹配该模式的所有索引的全部字段与字段映射。如果是基于时间的索引模式，还会有主时间字段 timeFieldName 和时间间隔 intervalName 两个字段。

field 数组中，每个元素是一个字段的情况，包括字段的 type, name, indexed, analyzed, doc_values, count, scripted，searchable，aggregationable，format，popularity 这些状态。

如果 scripted 为 true，那么这个元素就是通过 kibana 页面添加的脚本化字段，那么这条字段记录还会额外多几个内容：

- script: 记录实际 script 语句。
- lang: 在 Elasticsearch 的 datanode 上采用什么 lang-plugin 运行。默认是 painless。即 Elasticsearch 5 开始默认启用的 脚本引擎。可以在 management 页面上切换成 Lucene expression 等其他你的集群中可用的脚本语言。

timelion_sheet

包括有 timelion_chart_height，timelion_columns，timelion_interval，timelion_other_interval，timelion_rows，timelion_sheet 等字段

小贴士

在本书之前介绍 packetbeat 时提到的自带 dashboard 导入脚本，其实就是通过 `curl` 命令上传这些 JSON 到 kibana_index 索引里。

主页入口

我们先从启动 Kibana 的命令行程序入手，可以看到这是一个 shell 脚本。最终执行的是 `node src/cli serve` 命令。然后跟着就可以找到 `src/cli/serve` 程序，其中最重要的是加载了 `src/server/kbn_server.js` 。继续打开，可以看到它先后加载了 config, http, logging, plugin 和 uiExports。毫无疑问，其中重点是 http 和 uiExports 部分。

`http/index.js` 中，初始化了 Hapi.Server 对象，加载 hapi plugin，并声明了主要的 route。包括静态文件、模板文件、短地址跳转和主页默认跳转到 `/app/kibana` 。目前来说 Kibana 在服务器端主动做的事情还比较少。在我们不基于 Hapi 框架做二次开发的情况下，不用过于关注这期间 Kibana 做了什么。

下面进入 `src/ui/` 目录继续。

`src/ui/index.js` 中完成了更细节的各类 app 的加载和路由分配：

```
const uiExports = kbnServer.uiExports = new UiExports({
  urlBasePath: config.get('server.basePath')
});
for (let plugin of kbnServer.plugins) {
  uiExports.consumePlugin(plugin);
}

const bundles = kbnServer.bundles = new UiBundleCollection(bundl
erEnv, config.get('optimize.bundleFilter'));

for (let app of uiExports.getAllApps()) {
  bundles.addApp(app);
}
server.route({
  path: '/app/{id}',
  method: 'GET',
  handler: function (req, reply) {
    const id = req.params.id;
    const app = uiExports.apps.byId[id];
    if (!app) return reply(Boom.notFound('Unknown app ' + id));

    if (kbnServer.status.isGreen()) {
      return reply.renderApp(app);
    } else {
      return reply.renderStatusPage();
    }
  }
});
```

可以看到这里把所有的 app 都打包进了 bundle。这也是很多初次接触 Kibana 二次开发的新手很容易被绊倒的一点——改了一行代码怎么没生效？因为服务是优先使用 bundle 内容的，而不会每次都进到各源码目录执行。

如果确实在频繁修改代码的阶段，每次都等 bundle 确实太累了，可以看到上面代码段里有一个 `config.get('optimize.bundleFilter')` 。是的，其实 Kibana 支持在 config 中设定具体的 optimize 行为，但是官方文档上并没有介绍。最完整的配置项，见 `src/server/config/schema.js` 。前文说过，这是在启动 `kbn_server` 的时候最先加载的。

在 schema 中可以看到一个很可爱的配置：

```
optimize: _joi2['default'].object({
  enabled: _joi2['default'].boolean()['default'](true),
})
```

所以你只要在 `config/kibana.yml` 中加上这么一行配置就好了：`optimize.enabled: false`。

kibana app

从 Kibana 4.5 版开始，Kibana 框架和 Kibana App 做了一个剥离。现在，我们进到 Kibana App 里看看。路径在 `src/core_plugins/kibana`。

我们可以看到路径中有如下文件：

- common/
- index.js
- package.json
- public/
- server/

这是一个很显然的普通 nodejs 模块的结构。我们可以看看作为模块描述的 package.json 里写了啥：

```
{
  "name": "kibana",
  "version": "kibana"
}
```

非常有趣的 version。事实上这个写法的意思是本插件的版本号和 Kibana 框架的版本号保持一致。事实上所有 core_plugins 的版本号都写的是 kibana。

然后 index.js 中，调用 uiExports 完成了 app 注册。也这是之后我们自己开发新的 Kibana 应用时必须做的。我们下面摘主要段落分别看一下：

```
module.exports = function (kibana) {
  var kbnBaseUrl = '/app/kibana';
  return new kibana.Plugin({
    id: 'kibana',
    config: function config(Joi) {
      return Joi.object({
        enabled: Joi.boolean()['default'](true),
        defaultAppId: Joi.string()['default']('discover'),
        index: Joi.string()['default']('.kibana')
      })['default']();
    },
    uiExports: {
      app: {
         id: 'kibana',
         title: 'Kibana',
         listed: false,
         description: 'the kibana you know and love',
         main: 'plugins/kibana/kibana',
```

这是最基础的部分，注册成为一个 `kibana.Plugin` ，id 叫什么，config 配置有什么，标题叫什么，入口文件是哪个，具体是什么类型的 uiExports，一般常见的选择有：app、visType。这两者也是做 Kibana 二次开发最容易入手的地方。

```
         uses: ['visTypes', 'spyModes', 'fieldFormats', 'navbarE
xtensions', 'managementSections', 'devTools', 'docViews'],
         injectVars: function injectVars(server, options) {...}
      },
```

uses 和 injectVars 是可选的方式，可以在 `src/ui/ui_app.js` 中看到起作用。分别是指明下列模块已经加载过，以后就不用再加载了；以及声明需要注入浏览器的 JSON 变量。

```
        links: [{
          id: 'kibana:discover',
          title: 'Discover',
          order: -1003,
          url: kbnBaseUrl + '#/discover',
          description: 'interactively explore your data',
          icon: 'plugins/kibana/assets/discover.svg'
        }, {
          ...
        }],
      },
```

这里是一个特殊的地方，一般来说其他应用不会用到 links 类型的 uiExports。因为 Kibana 应用本身不用单一的左侧边栏切换，而是需要把自己内部的 Discover、Visualize、Dashboard、Management 功能放上去。所以定义里，把自己的 `listed` 给 false 了，而把这具体的四项通过 links 的方式，添加到侧边栏上。links 具体可配置的属性，见 `src/ui/ui_nav_link.js` 。这里就不细讲了。

```
      preInit: _asyncToGenerator(function* (server) {
          yield mkdirp(server.config().get('path.data'));
      }),
```

preInit 也是一个可选属性，如果有需要创建目录之类的要预先准备的操作，可以在这步完成。

```
    init: function init(server, options) {
      // uuid
      (0, _serverLibManage_uuid2['default'])(server);
      // routes
      (0, _serverRoutesApiIngest2['default'])(server);
      (0, _serverRoutesApiSearch2['default'])(server);
      (0, _serverRoutesApiSettings2['default'])(server);
      (0, _serverRoutesApiScripts2['default'])(server);

      server.expose('systemApi', systemApi);
    }
  });
}
```

init 是最后一步。我们看到 Kibana 应用的最后一步是继续加载了一些服务器端的 route 设置。比如这个 `_serverRoutesApiScripts2` ，具体代码是在 `src/core_plugins/kibana/server/routes/api/scripts/register_languages.js` 里：

```
server.route({
  path: '/api/kibana/scripts/languages',
  method: 'GET',
  handler: function handler(request, reply) {
    var callWithRequest = server.plugins.elasticsearch.callWithRequest;
    return callWithRequest(request, 'cluster.getSettings', {
      include_defaults: true,
      filter_path: '**.script.engine.*.inline'
    }).then(function (esResponse) {
      var langs = _lodash2['default'].get(esResponse, 'defaults.script.engine', {});
      var inlineLangs = _lodash2['default'].pick(langs, function (lang) {
        return lang.inline === 'true';
      });
      var supportedLangs = _lodash2['default'].omit(inlineLangs, 'mustache');
      return _lodash2['default'].keys(supportedLangs);
    }).then(reply)['catch'](function (error) {
      reply((0, _libHandle_es_error2['default'])(error));
    });
  }
});
```

我在之前 K4 源码解析中曾经讲过的一个二次开发场景 —— 切换脚本引擎支持。在 Elasticsearch 5.0 中，在 `/_cluster/settings` 接口里提供了具体的可用引擎细节，诸如一个个 `script.engine.painless.inline` 的列表。这样，就不必像之前那样明确知道自己可以用什么，然后硬改代码来支持了；而是可以通过这个接口数据，拿到集群实际支持什么引擎，当前默认是什么引擎等设置，直接在 Kibana 中使用。上面这段代码，就是提供了这个数据。

注意其中排除了 *mustache*，因为它只能做模板渲染，没法做字段值计算。

好了。应用注册完成，我们看到了 main 入口，那么去看看 main 入口的内容吧。打开 `src/core_plugins/kibana/public/kibana.js` 。主要如下：

```
import kibanaLogoUrl from 'ui/images/kibana.svg';
import 'ui/autoload/all';
import 'plugins/kibana/discover/index';
import 'plugins/kibana/visualize/index';
import 'plugins/kibana/dashboard/index';
import 'plugins/kibana/management/index';
import 'plugins/kibana/doc';
import 'plugins/kibana/dev_tools';
import 'ui/vislib';
import 'ui/agg_response';
import 'ui/agg_types';
import 'ui/timepicker';
import Notifier from 'ui/notify/notifier';
import 'leaflet';

routes.enable();

routes
.otherwise({
  redirectTo: `/${chrome.getInjected('kbnDefaultAppId', 'discover')}`
});

chrome
.setRootController('kibana', function ($scope, courier, config)
{
  $scope.$on('application.load', function () {
    courier.start();
  });
  ...
});
```

基本上通过这一串 import 就可以看到 Kibana 中最主要的各项功能了。

而对内部比较重要的则是这个 `ui/autoload/all` 。这里面其实是加载了 kibana 自定义的各种 angular module、directive 和 filter。像我们熟悉的 markdown、moment、auto_select、json_input、paginate、file_upload 等都在这里面加载。这些都是网页开发的通用工具，这里就不再介绍细节了，有兴趣的读者可以在 `src/ui/public/` 下找到对应文件。

设置 routes 的具体操作在加载的 `src/ui/public/routes/route_manager.js` 文件里，其中会调用 `sr/ui/public/index_patterns/route_setup/load_default.js` 中提供的 `addSetupWork` 方法，在未设置 default index pattern 的时候跳转 URL 到 `whenMissingRedirectTo` 页面。

```
  uiRoutes
  .addSetupWork(...)
  .afterWork(
    // success
    null,

    // failure
    function (err, kbnUrl) {
       let hasDefault = !(err instanceof NoDefaultIndexPattern);
       if (hasDefault || !whenMissingRedirectTo) throw err; // r
ethrow

       kbnUrl.change(whenMissingRedirectTo);
       if (!defaultRequiredToasts) defaultRequiredToasts = [];
       else defaultRequiredToasts.push(notify.error(err));
    }
  )
```

而这个 `whenMissingRedirectTo` 页面是在 kibana 应用的源码里写死的，见 `src/core_plugins/kibana/public/management/index.js`：

```
uiRoutes
.when('/management', {
      template: landingTemplate
});

require('ui/index_patterns/route_setup/load_default')({
      whenMissingRedirectTo: '/management/kibana/index'
});
```

在原先的版本中，routes 里面还会检查 Elasticsearch 的版本号，在 5.0 版里，这件事情从 kibana plugin 改到 elasticsearch plugin 里完成了。

courier 概述

kibana.js 的最后，控制器则会监听 `application.load` 事件，在页面加载完成的时候触发 `courier.start()` 函数。

`src/ui/public/courier/courier.js` 中定义了 **Courier** 类。**Courier** 是一个非常重要的东西，可以简单理解为 kibana 跟 ES 之间的一个 object mapper。简要的说，包括一下功能：

```
import DocSourceProvider from './data_source/doc_source';
...
function Courier() {
  var self = this;
  var DocSource = Private(DocSourceProvider);
  self.DocSource = DocSource;
  ...

  self.start = function () {
    searchLooper.start();
    docLooper.start();
    return this;
  };
  self.fetch = function () {
    fetch.fetchQueued(searchStrategy).then(function () {
      searchLooper.restart();
    });
  };
  self.started = function () {
    return searchLooper.started();
  };
  self.stop = function () {
    searchLooper.stop();
    return this;
  };
  self.createSource = function (type) {
    switch (type) {
    case 'doc':
      return new DocSource();
    case 'search':
      return new SearchSource();
```

```
          }
        };
        self.close = function () {
          searchLooper.stop();
          docLooper.stop();

          _.invoke(requestQueue, 'abort');

          if (requestQueue.length) {
            throw new Error('Aborting all pending requests failed.
  ');
          }
        };
```

从类的方法中可以看出，其实主要就是五个属性的控制：

- DocSource 和 SearchSource：继承自 `src/ui/public/courier/data_source/_abstract.js`，调用 `src/ui/public/courier/data_source/data_source/_doc_send_to_es.js` 完成跟 ES 数据的交互，用来做 savedObject 和 index_pattern 的读写：

```
      es[method](params)
      .then(function (resp) {
        if (resp.status === 409) throw new errors.VersionConflic
t(resp);

        doc._storeVersion(resp._version);
        doc.id(resp._id);

        var docFetchProm;
        if (method !== 'index') {
          docFetchProm = doc.fetch();
        } else {
          // we already know what the response will be
          docFetchProm = Promise.resolve({
            _id: resp._id,
            _index: params.index,
            _source: body,
            _type: params.type,
            _version: doc._getVersion(),
            found: true
          });
        }
```

这个 es 在是调用了 `src/ui/public/es.js` 里定义的 service，里面内容超级简单，就是加载官方的 elasticsearch.js 库，然后初始化一个最简的 esFactory 客户端，包括超时都设成了 0，把这个控制交给 server 端。

```
import 'elasticsearch-browser';
import _ from 'lodash';
import uiModules from 'ui/modules';

let es; // share the client amongst all apps
uiModules
  .get('kibana', ['elasticsearch', 'kibana/config'])
  .service('es', function (esFactory, esUrl, $q, esApiVersion, e
sRequestTimeout) {
    if (es) return es;
    es = esFactory({
```

```
      host: esUrl,
      log: 'info',
      requestTimeout: esRequestTimeout,
      apiVersion: esApiVersion,
      plugins: [function (Client, config) {
        // esFactory automatically injects the AngularConnector
to the config
        // https://github.com/elastic/elasticsearch-js/blob/mast
er/src/lib/connectors/angular.js
        _.class(CustomAngularConnector).inherits(config.connecti
onClass);
        function CustomAngularConnector(host, config) {
          CustomAngularConnector.Super.call(this, host, config);

          this.request = _.wrap(this.request, function (request,
 params, cb) {
          if (String(params.method).toUpperCase() === 'GET') {
            params.query = _.defaults({ _: Date.now()  }, params
.query);
          }
          return request.call(this, params, cb);
        });
      }
      config.connectionClass = CustomAngularConnector;
    }]
  });
  return es;
});
```

- searchLooper 和 docLooper：分别给 `Looper.start` 方法传递 searchStrategy 和 docStrategy，对应 ES 的 `/_msearch` 和 `/_mget` 请求。searchLooper 的实现如下：

```
import FetchProvider from '../fetch';
import SearchStrategyProvider from '../fetch/strategy/search';
import RequestQueueProvider from '../_request_queue';
import LooperProvider from './_looper';

export default function SearchLooperService(Private, Promise, No
tifier, $rootScope) {
    let fetch = Private(FetchProvider);
    let searchStrategy = Private(SearchStrategyProvider);
    let requestQueue = Private(RequestQueueProvider);

    let Looper = Private(LooperProvider);
    let searchLooper = new Looper(null, function () {
      $rootScope.$broadcast('courier:searchRefresh');
      return fetch.these(
        requestQueue.getInactive(searchStrategy)
      );
    });
    ...
```

这里的关键方法是 `fetch.these()` ，出自 `src/ui/public/courier/fetch/fetch_these.js` ，其中调用的 `src/ui/public/courier/fetch/call_client.js` 有如下一段代码：

```
        Promise.map(executable, function (req) {
          return Promise.try(req.getFetchParams, void 0, req)
          .then(function (fetchParams) {
            return (req.fetchParams = fetchParams);
          });
        })
        .then(function (reqsFetchParams) {
          return strategy.reqsFetchParamsToBody(reqsFetchParams);
        })
        .then(function (body) {
          return (esPromise = es[strategy.clientMethod]({ body }))
  ;
        })
        .then(function (clientResp) {
          return strategy.getResponses(clientResp);
        })
        .then(respond)
```

在这段代码中，我们可以看到 `strategy.reqsFetchParamsToBody()`，`strategy.getResponses()` 和 `strategy.clientMethod`，正是之前 searchLooper 和 docLooper 传递的对象属性。而最终发送请求，同样用的是前面解释过的 **es** 这个 service。

此外，Courier 还提供了自动刷新的控制功能：

```
      self.fetchInterval = function (ms) {
        searchLooper.ms(ms);
        return this;
      };
      ...
      $rootScope.$watchCollection('timefilter.refreshInterval',
function () {
        var refreshValue = _.get($rootScope, 'timefilter.refresh
Interval.value');
        var refreshPause = _.get($rootScope, 'timefilter.refresh
Interval.pause');
        if (_.isNumber(refreshValue) && !refreshPause) {
          self.fetchInterval(refreshValue);
        } else {
          self.fetchInterval(0);
        }
      });
```

路径记忆功能的实现

`src/ui/public/chrome/api/apps.js` 中，我们可以看到路径记忆功能是怎么实现的：

```
module.exports = function (chrome, internals) {
  internals.appUrlStore = internals.appUrlStore || window.sessio
nStorage;
  ...
  chrome.getLastUrlFor = function (appId) {
    return internals.appUrlStore.getItem(`appLastUrl:${appId}`);
  };
  chrome.setLastUrlFor = function (appId, url) {
    internals.appUrlStore.setItem(`appLastUrl:${appId}`, url);
  };
```

这里使用的 `sessionStorage` 是 HTML5 自带的新特性，这样，每次标签页切换的时候，都可以把 `$location.url` 保存下来。

搜索页

前文已经说到，kibana.js 中依次加载了各主要功能模块的入口。比如搜索页是 `src/core_plugins/kibana/public/discover/index.js` 。通过这个文件路径就可以猜到，有关搜索页的功能，代码应该都在 `src/core_plugins/kibana/public/discover/` 里了。这个目录下的文件有：

- _hit_sort_fn.js
- components/
- controllers/
- directives/
- index.html
- index.js
- partials/
- saved_searches/
- styles/

这也是一个比较标准的 angular 模块的目录结构了。一眼就能知道，controller、directive 等等分别应该进哪里去看。当然首先第一步还是看 index.js：

```
import 'plugins/kibana/discover/saved_searches/saved_searches';
import 'plugins/kibana/discover/directives/no_results';
import 'plugins/kibana/discover/directives/timechart';
import 'ui/collapsible_sidebar';
import 'plugins/kibana/discover/components/field_chooser/field_chooser';
import 'plugins/kibana/discover/controllers/discover';
import 'plugins/kibana/discover/styles/main.less';
import 'ui/doc_table/components/table_row';
import savedObjectRegistry from 'ui/saved_objects/saved_object_registry';

savedObjectRegistry.register(require('plugins/kibana/discover/saved_searches/saved_search_register'));
```

已存搜索、事件数趋势图、事件列表、字段列表，各自载入了。下面可以看一下这几个功能点的实现。

plugins/kibana/discover/saved_searches/saved_searches.js

- 定义 savedSearches 这个 angular service，用来操作 kbnIndex 索引里 search 这个类型下的数据；
- 加载了 `saved_searches/_saved_searches.js` 提供的 savedSearch 这个 angular factory，这里定义了一个搜索 (search) 在 `kbnIndex` 里的数据结构，包括 title, description, hits, column, sort, version 等字段(这部分内容，可以直接通过读取 Elasticsearch 中的索引内容看到，比阅读代码更直接，本章最后即专门介绍 `kbnIndex` 中的数据结构)，看的是不是有点眼熟？没错，这个 savedSearch 就是继承了上一节我们介绍的那个 courier 的 savedObject：

```
module.factory('SavedSearch', function (courier) {
  _.class(SavedSearch).inherits(courier.SavedObject);
  function SavedSearch(id) {
    ...
```

- 还加载并注册了 `plugins/kibana/management/saved_object_registry`，表示可以在 management 里修改这里的 savedSearches 对象。

plugins/kibana/discover/directives/timechart.js

- 加载 `ui/vislib`。
- 提供 discoverTimechart 这个 angular directive，监听 "data" 并调用 `vislib.Chart` 对象绘图。

vislib 是整个 Kibana 可视化的实现部分，下一节会更详细的讲述。

plugins/kibana/discover/components/field_chooser/field_chooser.js

- 提供 discFieldChooser 这个 angular directive，其中监听字段和事件的变化，并调用计算常用字段排行，

```
        $scope.$watchMulti([
          '[]fieldCounts',
          '[]columns',
          '[]hits'
        ], function (cur, prev) {
          var newHits = cur[2] !== prev[2];
          var fields = $scope.fields;
          var columns = $scope.columns || [];
          var fieldCounts = $scope.fieldCounts;

          if (!fields || newHits) {
            $scope.fields = fields = getFields();
          }

          _.chain(fields)
          .each(function (field) {
            field.displayOrder = _.indexOf(columns, field.name)
+ 1;
            field.display = !!field.displayOrder;
            field.rowCount = fieldCounts[field.name];
          })
          .sortBy(function (field) {
            return (field.count || 0) * -1;
          })
          .groupBy(function (field) {
            if (field.display) return 'selected';
            return field.count > 0 ? 'popular' : 'unpopular';
          })
          .tap(function (groups) {
            groups.selected = _.sortBy(groups.selected || [], 'd
isplayOrder');
            groups.popular = groups.popular || [];
            groups.unpopular = groups.unpopular || [];
            var extras = groups.popular.splice(config.get('field
s:popularLimit'));
            groups.unpopular = extras.concat(groups.unpopular);
          })
```

```
          .each(function (group, name) {
            $scope[name + 'Fields'] = _.sortBy(group, name === '
selected' ? 'display' : 'name');
          })
          .commit();

          $scope.fieldTypes = _.union([undefined], _.pluck(field
s, 'type'));
        });
```

监听 "data" 并调用 `$scope.details()` 方法，

提供 `$scope.vizLocation()` 方法。方法中，根据字段的类型不同，分别可能使用 `date_histogram` / `geohash_grid` / `terms` 聚合函数，创建可视化模型，然后带着当前页这些设定——前面说过，各 app 之间通过 globalState 共享状态，也就是 URL 中的 `?_a=(...)` 。各 app 会通过 `rison.decode($location.search()._a)` 和 `rison.encode($location.search()._a)` 设置和读取——跳转到 "/visualize/create" 页面，相当于是这三个常用聚合的快速可视化操作。

默认的 create 页的 rison 如下：

```
          return '#/visualize/create?' + $.param(_.assign($locat
ion.search(), {
            indexPattern: $scope.state.index,
            type: type,
            _a: rison.encode({
              filters: $scope.state.filters || [],
              query: $scope.state.query || undefined,
              vis: {
                type: type,
                aggs: [
                  agg,
                  {schema: 'metric', type: 'count', 'id': '2'}
                ]
              }
            })
          });
```

之前章节的 url 示例中，读者如果注意的话，会发现 id 是从 2 开始的，原因即在此。

- 加载 `plugins/kibana/discover/components/field_chooser/lib/field_calculator.js`，提供 `fieldCalculator.getFieldValueCounts()` 方法，在 `$scope.details()` 中读取被点击的字段值的情况。
- 加载 `plugins/kibana/discover/components/field_chooser/discover_field.js`，提供 discoverField 这个 angular directive，用于弹出浮层展示临时的 visualize(调用上一条提供的 `$scope.details()` 方法)，同时给被点击的字段加常用度；加载 `plugins/kibana/discover/components/field_chooser/lib/detail_views/string.html` 网页，用于浮层效果。网页中对 indexed 或 scripted 类型的字段，可以调用前面提到的 `vizLocation()` 方法。

```
        import detailsHtml from 'plugins/kibana/discover/components/field_chooser/lib/detail_views/string.html';
        $scope.toggleDetails = function (field, recompute) {
          if (_.isUndefined(field.details) || recompute) {
            // This is inherited from fieldChooser
            $scope.details(field, recompute);
            detailScope.$destroy();
            detailScope = $scope.$new();
            detailScope.warnings = getWarnings(field);

            detailsElem = $(detailsHtml);
            $compile(detailsElem)(detailScope);
            $elem.append(detailsElem).addClass('active');
          } else {
            delete field.details;
            detailsElem.remove();
            $elem.removeClass('active');
          }
        };
```

- 加载并渲染 `plugins/kibana/discover/components/field_chooser/field_chooser.`

`html` 网页。网页中使用了上一条提供的 discover-field。

plugins/kibana/discover/controllers/discover.js

加载了诸多 js，主要做了：

- 为 "/discover/:id?" 提供 route 并加载 `plugins/discover/index.html` 网页。
- 提供 discoverApp 这个 angular controller。
- 加载 `ui/vis.js` 并在 setupVisualization 函数中绘制 histogram 图。

```
      var visStateAggs = [
        {
          type: 'count',
          schema: 'metric'
        },
        {
          type: 'date_histogram',
          schema: 'segment',
          params: {
            field: $scope.opts.timefield,
            interval: $state.interval
          }
        }
      ];

      $scope.vis = new Vis($scope.indexPattern, {
        title: savedSearch.title,
        type: 'histogram',
        params: {
          addLegend: false,
          addTimeMarker: true
        },
        listeners: {
          click: function (e) {
            timefilter.time.from = moment(e.point.x);
            timefilter.time.to = moment(e.point.x + e.data.order
ed.interval);
            timefilter.time.mode = 'absolute';
          },
          brush: brushEvent
        },
        aggs: visStateAggs
      });

      $scope.searchSource.aggs(function () {
        $scope.vis.requesting();
        return $scope.vis.aggs.toDsl();
      });
```

visualize

index.js 中，首要当然是注册自己。此外，还加载两部分功能：`plugins/kibana/visualize/editor/*` 和 `plugins/kibana/visualize/wizard/wizard.js`。然后定义了一个 route，默认跳转 `/visualize` 到 `/visualize/step/1`。

editor

editor.js 中也定义了两个 route，分别是 `/visualize/create` 和 `/visualize/edit/:id`。然后还定义了一个controller，叫 `VisEditor`，对应的 HTML 是 `plugins/kibana/visualize/editor/editor.html`，其中用到两个 directive，分别是 `visualize` 和 `vis-editor-sidebar`。

其中 create 是先加载 `ui/registry/vis_types`，并检查 `$route.current.params.type` 是否存在，然后调用 `savedVisualizations.get($route.current.params)` 方法；而 edit 是直接调用 `savedVisualizations.get($route.current.params.id)`。

vis_types

实际注册了 `vis_types` 的地方包括：

- plugins/table_vis/index.js
- plugins/metric_vis/index.js
- plugins/markdown_vis/index.js
- plugins/kbn_vislib_vis_types/index.js

前三个是表单，最后一个是可视化图。内容如下：

```
import visTypes from 'ui/registry/vis_types';
visTypes.register(require('plugins/kbn_vislib_vis_types/histogra
m'));
visTypes.register(require('plugins/kbn_vislib_vis_types/line'));
visTypes.register(require('plugins/kbn_vislib_vis_types/pie'));
visTypes.register(require('plugins/kbn_vislib_vis_types/area'));
visTypes.register(require('plugins/kbn_vislib_vis_types/tile_map
'));
```

以 histogram 为例解释一下 visTypes。下面的实现较长，我们拆成三部分：

第一部分，加载并生成VislibVisType对象：

```
import VislibVisTypeVislibVisTypeProvider from 'ui/vislib_vis_ty
pe/vislib_vis_type';
import VisSchemasProvider from 'ui/vis/schemas';
import histogramTemplate from 'plugins/kbn_vislib_vis_types/edit
ors/histogram.html';

export default function HistogramVisType(Private) {
  const VislibVisType = Private(VislibVisTypeVislibVisTypeProvid
er);
  const Schemas = Private(VisSchemasProvider);

  return new VislibVisType({
    name: 'histogram',
    title: 'Vertical bar chart',
    icon: 'fa-bar-chart',
    description: 'The goto chart for oh-so-many needs. Great for
 time and non-time data. Stacked or grouped, ' +
    'exact numbers or percentages. If you are not sure which cha
rt you need, you could do worse than to start here.',
```

第二部分，histogram 可视化所接受的参数默认值以及对应的参数编辑页面：

```
        params: {
          defaults: {
            shareYAxis: true,
            addTooltip: true,
            addLegend: true,
            legendPosition: 'right',
            scale: 'linear',
            mode: 'stacked',
            times: [],
            addTimeMarker: false,
            defaultYExtents: false,
            setYExtents: false,
            yAxis: {}
          },
          scales: ['linear', 'log', 'square root'],
          modes: ['stacked', 'percentage', 'grouped'],
          editor: histogramTemplate
        },
```

第三部分，histogram 可视化能接受的 Schema。一般来说，metric 数值聚合肯定是 Y 轴；bucket 聚合肯定是 X 轴；而在此基础上，Kibana4 还可以让 bucket 有不同效果，也就是 Schema 里的 segment(默认), group 和 split。根据效果不同，这里是各有增减的，比如饼图就不会有 group。

```
      schemas: new Schemas([
        {
          group: 'metrics',
          name: 'metric',
          title: 'Y-Axis',
          min: 1,
          aggFilter: '!std_dev',
          defaults: [
            { schema: 'metric', type: 'count' }
          ]
        },
        {
          group: 'buckets',
          name: 'segment',
          title: 'X-Axis',
          min: 0,
          max: 1,
          aggFilter: '!geohash_grid'
        },
        {
          group: 'buckets',
          name: 'group',
          title: 'Split Bars',
          min: 0,
          max: 1,
          aggFilter: '!geohash_grid'
        },
        {
          group: 'buckets',
          name: 'split',
          title: 'Split Chart',
          min: 0,
          max: 1,
          aggFilter: '!geohash_grid'
        }
      ])
    });
  };
});
```

这里使用的 VislibVisType 类，继承自 `ui/vis/VisType.js` ，VisType.js 内容如下：

```
import VisSchemasProvider from 'ui/vis/schemas';

export default function VisTypeFactory(Private) {
  let VisTypeSchemas = Private(VisSchemasProvider);

function VisType(opts) {
  opts = opts || {};

    this.name = opts.name;
    this.title = opts.title;
    this.responseConverter = opts.responseConverter;
    this.hierarchicalData = opts.hierarchicalData || false;
    this.icon = opts.icon;
    this.description = opts.description;
    this.schemas = opts.schemas || new VisTypeSchemas();
    this.params = opts.params || {};
    this.requiresSearch = opts.requiresSearch == null ? true : o
pts.requiresSearch; // Default to true unless otherwise specifie
d
  }

  return VisType;
};
```

基本跟上面 histogram 的示例一致，注意这里面的 responseConverter 和 hierarchicalData，是给不同的 visType 做相应数据转换的。在实际的 VislibVisType 中，就有下面一段：

```
      if (this.responseConverter == null) {
        this.responseConverter = pointSeries;
      }
```

可见默认情况下，Kibana 是尝试把聚合结果转换成点线图数组的。

VislibVisType 中另一部分，则是扩展了一个自己的方法 createRenderbot，用来生成 VislibRenderbot 对象。这个类的实现在 `ui/vislib_vis_type/vislib_renderbot.js`，其中最关键的几行是：

```
import VislibVisTypeBuildChartDataProvider from 'ui/vislib_vis_t
ype/build_chart_data';
module.exports = function VislibRenderbotFactory(Private, $injec
tor) {
  let buildChartData = Private(VislibVisTypeBuildChartDataProvid
er);
  ...
    self.vislibVis = new vislib.Vis(self.$el[0], self.vislibPara
ms);
  ...
  VislibRenderbot.prototype.buildChartData = buildChartData;
  VislibRenderbot.prototype.render = function (esResponse) {
    this.chartData = this.buildChartData(esResponse);
    return AngularPromise.delay(1).then(() => {
      this.vislibVis.render(this.chartData, this.uiState);
    });
  };
```

也就是说，分为两部分，buildChartData 方法和 vislib.Vis 对象。

先来看 buildChartData 的实现：

```
import AggResponseIndexProvider from 'ui/agg_response/index';
...
    return function (esResponse) {
      let vis = this.vis;
      if (vis.isHierarchical()) {
        return aggResponse.hierarchical(vis, esResponse);
      }
      let tableGroup = aggResponse.tabify(vis, esResponse, {
        canSplit: true,
        asAggConfigResults: true
      });
      let converted = convertTableGroup(vis, tableGroup);
      if (!converted) {
        converted = { rows: []  };
      }
      converted.hits = esResponse.hits.total;
      return converted;
    };
    ....
    function convertTable(vis, table) {
      return vis.type.responseConverter(vis, table);
    }
```

又看到 responseConverter 和 hierarchical 两个熟悉的字眼了，不过这回是另一个对象的方法，那么我们继续跟踪下去，看看这个 aggResponse 类是怎么回事：

```
import AggResponseHierarchicalBuildHierarchicalDataProvider from
 'ui/agg_response/hierarchical/build_hierarchical_data';
import AggResponsePointSeriesPointSeriesProvider from 'ui/agg_re
sponse/point_series/point_series';
import AggResponseTabifyTabifyProvider from 'ui/agg_response/tab
ify/tabify';
import AggResponseGeoJsonGeoJsonProvider from 'ui/agg_response/g
eo_json/geo_json';

export default function NormalizeChartDataFactory(Private) {
  return {
    hierarchical: Private(AggResponseHierarchicalBuildHierarchic
alDataProvider),
    pointSeries: Private(AggResponsePointSeriesPointSeriesProvid
er),
    tabify: Private(AggResponseTabifyTabifyProvider),
    geoJson: Private(AggResponseGeoJsonGeoJsonProvider)
  };
};
```

然后我们看 vislib.Vis 对象，定义在 `ui/public/vislib/vis.js` 里。同时我们注意到，定义 vislib 这个服务的 `ui/public/vislib/index.js` 里，还导入了一个模块，叫 d3，没错，我们离真正的绘图越来越近了。

vis.js 中加载了 `ui/vislib/lib/handler/handler_types` 和 `ui/vislib/visualizations/vis_types` ：

```
import VislibLibHandlerHandlerTypesProvider from 'ui/vislib/lib/
handler/handler_types';
import VislibVisualizationsVisTypesProvider from 'ui/vislib/visu
alizations/vis_types';
```

chartTypes 用来定义图：

```
class Vis extends Events {
  constructor($el, config) {
    super(arguments);
    this.el = $el.get ? $el.get(0) : $el;
    this.binder = new Binder();
    this.ChartClass = chartTypes[config.type];
    this._attr = _.defaults({}, config || {}, {
      legendOpen: true
    });
```

接着是 handlerTypes 用来绘制图：

```
    render(data, uiState) {
      var chartType = this._attr.type;
      this.data = data;
      this.handler = handlerTypes[chartType](this) || handlerTyp
es.column(this);
      this._runWithoutResizeChecker('render');
    };


    _runWithoutResizeChecker(method) {
      this.resizeChecker.stopSchedule();
      this._runOnHandler(method);
      this.resizeChecker.saveSize();
      this.resizeChecker.saveDirty(false);
      this.resizeChecker.continueSchedule();
    };
    _runOnHandler = function (method) {
      this.handler[method]();
    };
```

`ui/vislib/lib/handler/handler_types` 中，根据不同的 vis_types，分别返回不同的处理对象，主要出自 `ui/vislib/lib/handler/types/point_series`，`ui/vislib/lib/handler/types/pie` 和 `ui/vislib/lib/handler/types/tile_map`。比如 histogram 就是 `pointSeries.column`。可以看到 point_series.js 中，对 column 是加上了 `zeroFill:true, expandLastBucket:true` 两个参数调用 `create()` 方法。

而 `create()` 方法里的 `new Handler()` 传递的，显然就是给 d3.js 的绘图参数。而 Handler 具体初始化和渲染过程，则在被加载的 `ui/vislib/lib/handler/handler.js` 中。`Handler.prototype.render` 中如下一段：

```
const selection = d3.select(this.el);
const chartSelection = selection..selectAll('.chart');
chartSelection.each(function (chartData) {
  const chart = new self.ChartClass(self, this, chartData);
  self.vis.activeEvents().forEach(function (event) {
    self.enable(event, chart);
  });

  binder.on(chart.events, 'rendered', () => {
    loadedCount++;
    if (loadedCount === chartSelection.length) {
      charts[0].events.emit('renderComplete');
    }
  });
  charts.push(chart);
  chart.render();
});
```

这里面的 `ChartClass()` 就是在 vislib.js 中加载了的 `ui/vislib/visualizations/vis_types` 。它会根据不同的 vis_types，分别返回不同的可视化对象，包括： `ui/vislib/visualizations/column_chart` , `ui/vislib/visualizations/pie_chart` , `ui/vislib/visualizations/line_chart` , `ui/vislib/visualizations/area_chart` 和 `ui/vislib/visualizations/tile_map` 。

这些对象都有同一个基类： `ui/vislib/visualizations/_chart` ，其中有这么一段：

```
    render() {
      const selection = d3.select(this.chartEl);

      selection.selectAll('*').remove();
      selection.call(this.draw());
    };
```

也就是说，各个可视化对象，只需要用 d3.js 或者其他绘图库，完成自己的 **draw()** 函数，就可以了！

draw 函数的实现一般格式，就像下面这段出自 LineChart 的代码：

```
    draw() {
      const self = this;
      const $elem = $(this.chartEl);
      const margin = this._attr.margin;
      const elWidth = this._attr.width = $elem.width();
      const elHeight = this._attr.height = $elem.height();
      const scaleType = this.handler.yAxis.getScaleType();
      const yScale = this.handler.yAxis.yScale;
      const xScale = this.handler.xAxis.xScale;
      const minWidth = 20;
      const minHeight = 20;
      const startLineX = 0;
      const lineStrokeWidth = 1;
      const addTimeMarker = this._attr.addTimeMarker;
      const times = this._attr.times || [];
      let timeMarker;

      return function (selection) {
        selection.each(function (data) {
          const el = this;
          const layers = data.series.map(function mapSeries(d) {
            const label = d.label;
            return d.values.map(function mapValues(e, i) {
              return {
                _input: e,
                label: label,
                x: self._attr.xValue.call(d.values, e, i),
```

```
                y: self._attr.yValue.call(d.values, e, i)
              };
            });
          });
          const width = elWidth - margin.left - margin.right;
          const height = elHeight - margin.top - margin.bottom;
          const div = d3.select(el);
          const svg = div.append('svg')
          .attr('width', width + margin.left + margin.right)
          .attr('height', height + margin.top + margin.bottom)
          .append('g')
          .attr('transform', 'translate(' + margin.left + ',' +
margin.top + ')');

          // 处理 data 到 svg 上
          ...
          self.events.emit('rendered', {
            chart: data
          });
          return svg;
        });
      };
    };
```

当然，为了代码逻辑，有些比较复杂的绘制，还是会继续拆分成其他文件的。比如 leaflet 地图，就是在 `ui/vislib/visualizations/tile_map` 里加载的 `ui/vislib/visualizations/_map.js` 完成。

从数据到 d3 渲染，要经过的主要流程就是这样。如果打算自己亲手扩展一个新的可视化方案的读者，可以具体参考我实现的 sankey 图：https://github.com/chenryn/kibana4/commit/4e0bcbeb4c8fd94807c3a0b1df2ac6f56634f9a5

savedVisualizations

这个类在 `core_plugins/kibana/public/visualize/saved_visualizations/saved_visualizations.js` 里定义。其中分三步，加载

`core_plugins/kibana/public/visualize/saved_visualizations/_saved_vis` ，注册到 `plugins/kibana/management/saved_object_registry` ，以及定义一个 angular service 叫 `savedVisualizations` 。

`plugins/kibana/visualize/saved_visualizations/_saved_vis` 里是定义一个 angular factory 叫 `SavedVis` 。这个类继承自 **courier.SavedObject**，主要有 `_getLinkedSavedSearch` 方法调用 `savedSearches` 获取在 discover 中保存的 search 对象，以及 visState 属性。该属性保存了 visualize 定义的 JSON 数据。

savedVisualizations 里主要就是初始化 SavedVis 对象，以及提供了一个 find 搜索方法。整个实现和上一节讲的 savedSearches 基本一样，就不再讲了。

Visualize

这个 directive 在 `ui/visualize/visualize.js` 中定义。而我们可以上拉看到的请求、响应、表格、性能数据，则使用的是 `ui/visualize/spy/spy.js` 中定义的另一个 directive `visualizeSpy` 。

visualize.html 上定义了一个普通的 div，其 class 为 **visualize-chart**，在 visualize.js 中，通过 `getter('.visualize-chart')` 方法获取 div 元素：

```
        function getter(selector) {
          return function () {
            let $sel = $el.find(selector);
            if ($sel.size()) return $sel;
          };
        }
        let getVisEl = getter('.visualize-chart');
```

然后创建一个 renderbot：

```
        $scope.$watch('vis', prereq(function (vis, oldVis) {
          let $visEl = getVisEl();
          if (!$visEl) return;

          if (!attr.editableVis) {
            $scope.editableVis = vis;
          }

          if (oldVis) $scope.renderbot = null;
          if (vis) $scope.renderbot = vis.type.createRenderbot(v
is, $visEl);
        }));
```

最后在 searchSource 对象变化，即有新的搜索响应返回时，完成渲染：

```
        $scope.$watch('searchSource', prereq(function (searchSou
rce) {
          if (!searchSource || attr.esResp) return;
          searchSource.onResults().then(function onResults(resp)
 {
            if ($scope.searchSource !== searchSource) return;

            $scope.esResp = resp;

            return searchSource.onResults().then(onResults);
          }).catch(notify.fatal);
          searchSource.onError(notify.error).catch(notify.fatal)
;
        }));
        $scope.$watch('esResp', prereq(function (resp, prevResp)
 {
          if (!resp) return;
          $scope.renderbot.render(resp);
        }));
```

VisEditorSidebar

这个 directive 在 `plugins/kibana/visualize/editor/sidebar.js` 中定义。对应的 HTML 是 `plugins/kibana/visualize/editor/sidebar.html`，其中又用到两个 directive，分别是 `vis-editor-agg-group` 和 `vis-editor-vis-options`。它们分别有 sidebar.js 加载的 `plugins/kibana/visualize/editor/agg_group` 和 `plugins/kibana/visualize/editor/vis_options` 提供。然后继续 HTML -> directive 下去，基本上 `plugins/kibana/visualize/editor/` 目录下那堆 `agg*.js` 和 `agg*.html` 都是做这个用的。

其中比较有意思的，应该算是 `agg_add.js`。我们都知道，K4 最大的特点就是可以层叠子聚合，这个操作就是在这里完成的：

```
import VisAggConfigProvider from 'ui/vis/agg_config';
import uiModules from 'ui/modules';
import aggAddTemplate from 'plugins/kibana/visualize/editor/agg_
add.html';

uiModules
.get('kibana')
.directive('visEditorAggAdd', function (Private) {
  const AggConfig = Private(VisAggConfigProvider);

  return {
    restrict: 'E',
    template: aggAddTemplate,
    controllerAs: 'add',
    controller: function ($scope) {
      const self = this;

      self.form = false;
      self.submit = function (schema) {
        self.form = false;

        const aggConfig = new AggConfig($scope.vis, {
          schema: schema
        });
        aggConfig.brandNew = true;
        $scope.vis.aggs.push(aggConfig);
      };
    }
  };
});
```

另一个比较重要的是 `core_plugins/kibana/public/visualize/editor/agg_params.js`。其中加载了 `ui/agg_types/index.js`，又监听了 "agg.type" 变量，也就是实现了选择不同的 agg_types 时，提供不同的 agg_params 选项。比方说，选择 date_histogram，字段就只能是 `@timestamp` 这种 date 类型的字段。

`ui/agg_types/index.js` 中定义了所有可选 agg_types 的类。其中 metrics 包括：count, avg, sum, min, max, std_deviation, cardinality, percentiles, percentile_rank，具体实现分别存在 `ui/agg_types/metrics/` 目录下的同名*.js*文件里；buckets 包括：date_histogram, histogram, range, date_range, ip_range, terms, filters, significant_terms, geo_hash，具体实现分别存在 `ui/agg_types/buckets/` 目录下的同名*.js*文件里。

这些类定义中，都有比较类似的格式，其中 params 数组的第一个元素，都是类似这样：

```
{
  name: 'field',
  filterFieldTypes: 'string'
}
```

terms.js 里还多了一行 `scriptable: true`，而且 filterFieldTypes 是数组。

```
        {
          name: 'field',
          scriptable: true,
          filterFieldTypes: ['number', 'boolean', 'date', 'ip',
 'string']
        }
```

这个 `filterFieldTypes` 在 `ui/vis/_agg_config.js` 中，通过 `fieldTypeFilter(this.vis.indexPattern.fields, fieldParam.filterFieldTypes);` 得到可选字段列表。fieldTypeFilter 的具体实现在 `filters/filed_type.js` 中。

wizard

wizard.js 中提供两个 route 和对应的 controller。分别是 `/visualize/step/1` 对应 `VisualizeWizardStep1`，`/visualize/step/2` 对应 `VisualizeWizardStep2`。这两个的最终结果，都是跳转到 `/visualize/create?type=*` 下。

dashboard

`plugins/kibana/public/dashboard/index.js` 结构跟 visualize 类似，设置两个调用 `savedDashboards.get()` 方法的 routes，提供一个叫 dashboard-app 的 directive。

savedDashboards 由 `plugins/kibana/public/dashboard/services/saved_dashboard.js` 提供，调用 es.search 获取数据，生成 savedDashboard 对象，这个对象同样也是继承 savedObject，主要内容是 `panelsJSON` 数组字段。实现如下：

```
module.factory('SavedDashboard', function (courier) {
  _.class(SavedDashboard).inherits(courier.SavedObject);
  function SavedDashboard(id) {
    SavedDashboard.Super.call(this, {
      type: SavedDashboard.type,
      mapping: SavedDashboard.mapping,
      searchSource: SavedDashboard.searchsource,
      id: id,
      defaults: {
        title: 'New Dashboard',
        hits: 0,
        description: '',
        panelsJSON: '[]',
        optionsJSON: angular.toJson({
          darkTheme: config.get('dashboard:defaultDarkTheme')
        }),
        uiStateJSON: '{}',
        version: 1,
        timeRestore: false,
        timeTo: undefined,
        timeFrom: undefined,
        refreshInterval: undefined
      },
      clearSavedIndexPattern: true
    });
  }
```

可以注意到，这个 panelsJSON 是一个字符串，这跟之前 kbnIndex 提到的是一致的。

dashboard-app 中，最重要的功能，是监听搜索框和过滤条件的变更，我们可以看到 init 函数中有下面这段：

```
        function updateQueryOnRootSource() {
          var filters = queryFilter.getFilters();
          if ($state.query) {
            dash.searchSource.set('filter', _.union(filters, [{
              query: $state.query
            }]));
          } else {
            dash.searchSource.set('filter', filters);
          }
        }

        $scope.$listen(queryFilter, 'update', function () {
          updateQueryOnRootSource();
          $state.save();
        });
```

在 index.html 里，实际承载面板的，是下面这行：

```
  <dashboard-grid></dashboard-grid>
```

这也是一个 angular directive，通过加载 `plugins/kibana/public/dashboard/directives/grid.js` 引入的。其中添加面板相关的代码有两部分：

```
        $scope.$watchCollection('state.panels', function (pane
ls) {
            const currentPanels = gridster.$widgets.toArray().ma
p(function (el) {
              return getPanelFor(el);
            });
            const removed = _.difference(currentPanels, panels);
            const added = _.difference(panels, currentPanels);
            if (added.length) added.forEach(addPanel);
            ...
```

这段用来监听 `$state.panels` 数组，一旦有新增面板，调用 `addPanel` 函数。同理也有删除面板的，这里就不重复贴了。

而 addPanel 函数的实现大致如下：

```
        var addPanel = function (panel) {
          _.defaults(panel, {
            size_x: 3,
            size_y: 2
          });
          ...
          panel.$scope = $scope.$new();
          panel.$scope.panel = panel;
          panel.$el = $compile('<li><dashboard-panel></li>')(pan
el.$scope);
          gridster.add_widget(panel.$el, panel.size_x, panel.siz
e_y, panel.col, panel.row);
          ...
        };
```

这里即验证了之前 kbnIndex 小节中讲的 gridster 默认大小，又引入了一个新的 directive，叫 dashboard-panel。

dashboard-panel 在 `plugins/kibana/public/dashboard/components/panel/panel.js` 中实现，其中使用了

`plugins/kibana/public/dashboard/components/panel/panel.html` 页面。页面最后是这么一段：

```
<visualize ng-switch-when="visualization"
    vis="savedObj.vis"
    search-source="savedObj.searchSource"
    class="panel-content">
 </visualize>

 <doc-table ng-switch-when="search"
    search-source="savedObj.searchSource"
    sorting="panel.sort"
    columns="panel.columns"
    class="panel-content"
    filter="filter">
 </doc-table>
```

这里使用的 savedObj 对象，来自 `plugins/kibana/public/dashboard/components/panel/lib/load_panel.js` 获取的 savedSearch 或者 savedVisualization。获得的对象，以 savedVisualization 为例：

```
define(function (require) {
  return function visualizationLoader(savedVisualizations, Priva
te) { // Inject services here
    return function (panel, $scope) {
      return savedVisualizations.get(panel.id)
        .then(function (savedVis) {
          savedVis.vis.listeners.click = filterBarClickHandler($
scope.state);
          savedVis.vis.listeners.brush = brushEvent;

          return {
            savedObj: savedVis,
            panel: panel,
            editUrl: savedVisualizations.urlFor(panel.id)
          };
        });
    };
  };
});
```

而 visualize 和 doc-table 两个 directive。这两个正是之前在 visualize 和 discover 插件解析里提到过的，在 `components/` 底下实现。

Kibana 插件

Kibana 从 4.2 以后，引入了完善的插件化机制。目前分为 app，vistype，fieldformatter、spymode 等多种插件类型。原先意义上的 Kibana 现在已经变成了 Kibana 插件框架下的一个默认 app 类型插件。

本节用以讲述 Kibana 插件的安装使用和定制开发。

部署命令

安装 Kibana 插件有两种方式：

1. 通过 Elastic.co 公司的下载地址：

```
bin/kibana_plugin --install <org>/<package>/<version>
```

version 是可选项。这种方式目前适用于官方插件，比如：

```
bin/kibana_plugin -i elasticsearch/marvel/latest
bin/kibana_plugin -i elastic/timelion
```

1. 通过 zip 压缩包：

支持本地和远程 HTTP 下载两种，比如：

```
bin/kibana_plugin --install sense -u file:///tmp/sense-2.0.0-bet
a1.tar.gz
bin/kibana_plugin -i heatmap -u https://github.com/stormpython/h
eatmap/archive/master.zip
bin/kibana_plugin -i kibi_timeline_vis -u https://github.com/sir
ensolutions/kibi_timeline_vis/raw/0.1.2/target/kibi_timeline_vis
-0.1.2.zip
bin/kibana_plugin -i oauth2 -u https://github.com/trevan/oauth2/
releases/download/0.1.0/oauth2-0.1.0.zip
```

目前已知的 Kibana Plugin 列表见官方 WIKI：https://github.com/elastic/kibana/wiki/Known-Plugins

注意：kibana 目前版本变动较大，不一定所有插件都可以成功使用

查看与切换

插件安装完成后，可以在 Kibana 页面上通过 app switcher 界面切换。界面如下：



默认插件

除了 Kibana 本身以外，其实还有一些其他默认插件，这些插件本身在 app switcher 页面上是隐藏的，但是可以通过 url 直接访问到，或者通过修改插件的 `index.js` 配置项让它显示出来。

这些隐藏的默认插件中，最有可能被用到的，是 statusPage 插件。

我们可以通过 `http://localhost:5601/status` 地址访问这个插件的页面：

Status: Green tsathoggua

Heap Total (MB)	118.91	Heap Used (MB)	96.80	Load	2.31, 2.52, 2.31
Response Time Avg (ms)	10.37	Response Time Max (ms)	51.40	Requests Per Second	0.60

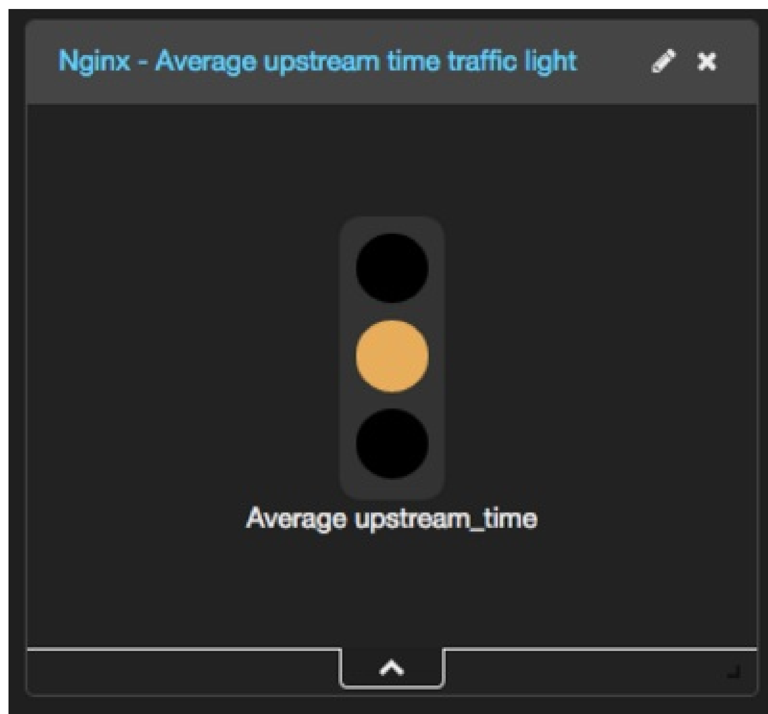
Status Breakdown

ID	Status
ui settings	✔ Ready
plugin:kibana@1.0.0	✔ Ready
plugin:elasticsearch@1.0.0	✔ Kibana index ready
plugin:timelion@5.0.0-alpha4	✔ Ready
plugin:console@1.0.0	✔ Ready
plugin:kbn_doc_views@1.0.0	✔ Ready
plugin:kbn_vislib_vis_types@1.0.0	✔ Ready
plugin:markdown_vis@1.0.0	✔ Ready
plugin:metric_vis@1.0.0	✔ Ready
plugin:spy_modes@1.0.0	✔ Ready
plugin:status_page@1.0.0	✔ Ready
plugin:table_vis@1.0.0	✔ Ready

页面会显示 Kibana 的运行状态。包括 nodejs 的内存使用、负载、响应性能，以及各插件的加载情况。

Kibana4 可视化插件开发

上一节，我们看到了一个完整的 Kibana 插件的官方用例。一般来说，我们不太会需要自己从头到尾写一个 angular app 出来。最常见的情况，应该是在 Kibana 功能的基础上做一定的二次开发和扩展。其中，可视化效果应该是重中之重。本节，以一个红绿灯效果，演示如何开发一个 Kibana 可视化插件。



插件目录生成

Kibana 开发组提供了一个简单的工具，辅助我们生成一个 Kibana 插件的目录结构：

```
npm install -g yo
npm install -g generator-kibana-plugin
mkdir traffic_light_vis
cd traffic_light_vis
yo kibana-plugin
```

但是这个是针对完整的 app 扩展的，很多目录对于可视化插件来说，并没有用。所以，我们可以自己手动创建目录：

```
mkdir -p traffic_light_vis/public
cd traffic_light_vis
touch index.js package.json
cd public
touch traffic_light_vis.html traffic_light_vis.js traffic_light_
vis.less traffic_light_vis_controller.js traffic_light_vis_param
s.html
```

其中 `index.js` 内容如下：

```
'use strict';
module.exports = function (kibana) {
    return new kibana.Plugin({
        uiExports: {
            visTypes: ['plugins/traffic_light_vis/traffic_light_
vis']
        }
    });
};
```

`package.json` 内容如下：

```
{
    "name": "traffic_light_vis",
    "version": "0.1.0",
    "description": "An awesome Kibana plugin for red/yellow/gree
n status visualize"
}
```

这两个基础文件格式都比较固定，除了改个名就基本 OK 了。

然后我们看最关键的可视化对象定义 `public/traffic_light_vis.js` 内容：

```
define(function (require) {
  // 加载样式表
  require('plugins/traffic_light_vis/traffic_light_vis.less');
  // 加载控制器程序
```

```
  require('plugins/traffic_light_vis/traffic_light_vis_controlle
r');
  // 注册到 vis_types
  require('ui/registry/vis_types').register(TrafficLightVisProvi
der);

  function TrafficLightVisProvider(Private) {
    // TemplateVisType 基类，适用于基础的 metric 和数据表格式的可视化定
义。实际上，Kibana4 的 metric_vis 和 table_vis 就继承自这个，
    // Kibana4 还有另一个基类叫 VisLibVisType，其他使用 D3.js 做可视化
的，继承这个。
    var TemplateVisType = Private(require('ui/template_vis_type/
TemplateVisType'));
    var Schemas = Private(require('ui/Vis/Schemas'));

    // 模板化的 visType 对象定义，用来配置和展示
    return new TemplateVisType({
      name: 'traffic_light',
      // 显示在 visualize 选择列表里的名称，描述，小图标
      title: 'Traffic Light',
      description: 'Great for one-glance status readings, the tr
affic light visualization expresses in green / yellow / red the
position of a single value in relation to low and high threshold
s.',
      icon: 'fa-car',
      // 可视化效果模板页面
      template: require('plugins/traffic_light_vis/traffic_light
_vis.html'),
      params: {
        defaults: {
          fontSize: 60
        },
        // 编辑参数的页面
        editor: require('plugins/traffic_light_vis/traffic_light
_vis_params.html')
      },
      // 在编辑页面上可以选择的 aggregation 类型。
      schemas: new Schemas([
        {
          group: 'metrics',
```

```
          name: 'metric',
          title: 'Metric',
          min: 1,
          defaults: [
            { type: 'count', schema: 'metric' }
          ]
        }
      ])
    });
  }

  // export the provider so that the visType can be required wit
h Private()
  return TrafficLightVisProvider;
});
```

然后就是可视化效果的模板页面了，`traffic_light_vis.html` 毫无疑问也是一个 angular 风格的：

```
<div ng-controller="TrafficLightVisController" class="traffic-light-vis">
    <div class="metric-container" ng-repeat="metric in metrics">
        <div class="traffic-light-container" ng-style="{'width': vis.params.width+'px', 'height': (2.68 * vis.params.width)+'px' }">
            <div class="traffic-light">
                <div class="light red" ng-class="{'on': (!vis.params.invertScale && metric.value <= vis.params.redThreshold) || (vis.params.invertScale && metric.value >= vis.params.redThreshold) }"></div>
                <div class="light yellow" ng-class="{'on': (!vis.params.invertScale && metric.value > vis.params.redThreshold && metric.value < vis.params.greenThreshold) || (vis.params.invertScale && metric.value < vis.params.redThreshold && metric.value > vis.params.greenThreshold) }"></div>
                <div class="light green" ng-class="{'on': (!vis.params.invertScale && metric.value >= vis.params.greenThreshold) || (vis.params.invertScale && metric.value <= vis.params.greenThreshold) }"></div>
            </div>
        </div>
        <div>{{metric.label}}</div>
    </div>
</div>
```

这里可以看到：

1. 把 div 绑定到了 `TrafficLightVisController` 控制器上，这个也是之前在 js 里已经加载过的。
2. 通过 `ng-repeat` 循环展示不同的 metric，也就是说模板渲染的时候，收到的是一个 `metrics` 数组。这个来源当然是在控制器里。
3. 然后具体的数据判断，即什么灯亮什么灯灭，通过了 `vis.params.*` 的运算判断。这些变量当然是在编辑页面里设置的。

所以下一步看编辑页面 `traffic_light_vis_params.html` ：

```
<div class="form-group">
  <label>Traffic light width - {{ vis.params.width }}px</label>
  <input type="range" ng-model="vis.params.width" class="form-co
ntrol" min="30" max="120"/>
</div>
<div class="form-group">
  <label>Red threshold <span ng-bind-template="({{!vis.params.in
vertScale ? 'below':'above'}} this value will be red)"></span></
label>
  <input type="number" ng-model="vis.params.redThreshold" class=
"form-control"/>
</div>
<div class="form-group">
  <label>Green threshold <span ng-bind-template="({{!vis.params.
invertScale ? 'above':'below'}} this value will be green)"></span
></label>
  <input type="number" ng-model="vis.params.greenThreshold" class
="form-control"/>
</div>
<div class="form-group">
  <label>
    <input type="checkbox" ng-model="vis.params.invertScale">
    Invert scale
  </label>
</div>
```

内容很简单，就是通过 `ng-model` 设置绑定变量，跟之前 HTML 里的联动。

最后一步，看控制器 `traffic_light_vis_controller.js` ：

```
define(function (require) {

  var module = require('ui/modules').get('kibana/traffic_light_v
is', ['kibana']);

  module.controller('TrafficLightVisController', function ($scop
e, Private) {
    var tabifyAggResponse = Private(require('ui/agg_response/tab
ify/tabify'));

    var metrics = $scope.metrics = [];

    $scope.processTableGroups = function (tableGroups) {
      tableGroups.tables.forEach(function (table) {
        table.columns.forEach(function (column, i) {
          metrics.push({
            label: column.title,
            value: table.rows[0][i]
          });
        });
      });
    };

    $scope.$watch('esResponse', function (resp) {
      if (resp) {
        metrics.length = 0;
        $scope.processTableGroups(tabifyAggResponse($scope.vis,
resp));
      }
    });
  });
});
```

要点在：

1. `$scope.$watch('esResponse', function(resp){})` 监听整个页面的请求响应，在有新数据过来的时候更新页面效果；
2. `agg_response/tabify/tabify` 把响应结果转换成二维表格形式。

最后加上一段样式表，这里就不贴了，见：https://github.com/logzio/kibana-visualizations/blob/master/traffic_light_vis/traffic_light_vis.less。

本节介绍的示例，出自 logz.io 官方博客和对应的 github 开源项目。logz.io 是基于 Kibana4.1 写的插件。我这里修正成了基于最新 Kibana4.3 的实现。

Kibana 服务器端插件开发

上一节介绍了如何给 Kibana 开发浏览器端的可视化插件。新版 Kibana 跟 Kibana3 比，最大的一个变化是有了独立的 node.js 服务器端。那么同样的，也就有了服务器端的 Kibana 插件。最明显的一个场景：我们可以在 node.js 里跑定时器做 Elasticsearch 的告警逻辑了！

本节示例一个最基础的 Kibana 告警插件开发。只演示基础的定时器和 Kibana 插件规范，实际运用中，肯定还涉及历史记录，告警项配置更新等。请读者不要直接 copy-paste。

首先，我们尽量沿袭 Elastic 官方的 watcher 产品的告警配置设计。也新建一个索引，里面是具体的配置内容：

```
# curl -XPUT http://127.0.0.1:9200/watcher/watch/error_status -d
'
{
  "trigger": {
    "schedule" : { "interval" : "60"  }
  },
  "input" : {
    "search" : {
      "request" : {
        "indices" : [ "<logstash-{now/d}>", "<logstash-{now/d-1d
}>"  ],
        "body" : {
          "query" : {
            "filtered" : {
              "query" : { "match" : { "host" : "MacBook-Pro"  }
},
              "filter" : { "range" : { "@timestamp" : { "from" :
 "now-5m"  } } }
            }
          }
        }
      }
    }
  },
```

```
  "condition" : {
    "script" : {
      "script" : "payload.hits.total > 0"
    }
  },
  "transform" : {
    "search" : {
      "request" : {
        "indices" : [ "<logstash-{now/d}>", "<logstash-{now/d-1d
}>"  ],
        "body" : {
          "query" : {
            "filtered" : {
              "query" : { "match" : { "host" : "MacBook-Pro"  }
},
              "filter" : { "range" : { "@timestamp" : { "from" :
 "now-5m"  } } }
            }
          },
          "aggs" : {
            "topn" : {
              "terms" : {
                "field" : "path.raw"
              }
            }
          }
        }
      }
    }
  },
  "actions" : {
    "email_admin" : {
    "throttle_period" : "15m",
    "email" : {
      "to" : "admin@domain",
      "subject" : "Found {{payload.hits.total}} Error Events",
      "priority" : "high",
      "body" : "Top10 paths:\n{{#payload.aggregations.topn.bucke
ts}}\t{{key}} {{doc_count}}\n{{/payload.aggregations.topn.bucket
s}}"
```

```
      }
    }
  }
}'
```

我们可以看到，跟原版的相比，只改动了很小的一些地方：

1. 为了简便，`interval` 固定写数值，没带 `s/m/d/H` 之类的单位；
2. `condition` 里直接使用了 JavaScript，这点也是 ES 2.x 的 mapping 要求跟 watcher 本身有冲突的一个地方：watcher的 `"ctx.payload.hits.total" : { "gt" : 0 }` 这种写法，如果是普通索引，会因为字段名里带 `.` 直接写入失败的；
3. 因为是在 Kibana 里面运行，所以从 ES 拿到的只有 payload(也就是查询响应)，所以把里面的 `ctx.` 都删掉了。

好，然后创建插件：

```
cd kibana-4.3.0-darwin-x64/src/plugins
mkdir alert
```

在自定义插件目录底下创建 `package.json` 描述：

```
{
  "name": "alert",
  "version": "0.0.1"
}
```

以及最终的 `index.js` 代码：

```
'use strict';
module.exports = function (kibana) {
  var later = require('later');
  var _ = require('lodash');
  var mustache = require('mustache');

  return new kibana.Plugin({
    init: function init(server) {
      var client = server.plugins.elasticsearch.client;
```

```
      var sched = later.parse.text('every 10 minute');
      later.setInterval(doalert, sched);
      function doalert() {
        getCount().then(function(resp){
          getWatcher(resp.count).then(function(resp){
            _.each(resp.hits.hits, function(hit){
              var watch = hit._source;
              var every = watch.trigger.schedule.interval;
              var watchSched = later.parse.recur().every(every).
second();
              var wt = later.setInterval(watching, watchSched);
              function watching() {
                var request = watch.input.search.request;
                var condition = watch.condition.script.script;
                var transform = watch.transform.search.request;
                var actions = watch.actions;
                client.search(request).then(function(payload){
                  var ret = eval(condition);
                  if (ret) {
                    client.search(transform).then(function(paylo
ad) {
                      _.each(_.values(actions), function(action)
{
                        if(_.has(action, 'email')) {
                          var subject = mustache.render(action.e
mail.subject, {"payload":payload});
                          var body = mustache.render(action.emai
l.body, {"payload":payload});
                          console.log(subject, body);
                        }
                      });
                    });
                  }
                });
              }
            });
          });
        });
      }
      function getCount() {
```

```
        return client.count({
          index:'watcher',
          type:"watch"
        });
      }
      function getWatcher(count) {
        return client.search({
          index:'watcher',
          type:"watch",
          size:count
        });
      }
    }
  });
};
```

其中用到了两个 npm 模块，later 模块用来实现定时器和 crontab 文本解析，mustache 模块用来渲染邮件内容模板，这也是 watcher 本身采用的渲染模块。

需要安装一下：

```
npm install later
npm install mustache
```

然后运行 `./bin/kibana` ，就可以看到终端上除了原有的内容以外，还会定期输出 alert 的 email 内容了。

要点解释

这个极简示例中，主要有两段：

1. 注册为插件

```
module.exports = function (kibana) {
  return new kibana.Plugin({
    init: function init(server) {
```

注意上一节的可视化插件，这块是：

```
module.exports = function (kibana) {
  return new kibana.Plugin({
    uiExports: {
      visTypes: [
```

1. 引用 ES client

```
    init: function init(server) {
      var client = server.plugins.elasticsearch.client;
```

这里通过调用 `server.plugins` 来直接引用 Kibana 里其他插件里的对象。这样，alert 插件就可以跟其他功能共用同一个 ES client，免去单独配置自己的 ES 设置项和新开网络连接的资源消耗。

本节代码后续优化改进，见：https://github.com/chenryn/kaae。项目中还附带有一个 spy 式插件，有兴趣的读者可以继续学习 spy 这类不太常见的插件扩展的用法。

app 开发

前面两节，我们分别看过了如何开发可视化部分和服务器端部分。现在，我们把这两头综合起来，做一个可以在 Kibana 菜单栏上切换使用的，完整的 app。就像 Kibana 5 默认分发的 timelion 和 console 那样。

当然我们这里不会真的特意搞一个很复杂的可视化应用。我们只做一个 Elasticsearch 状态展示页面就好了。这个方式正好可以串联从前到后的请求、展示部分。

app 模块的 index.js 结构

我们已经讲过了在 index.js 中如何使用 uiExports.visType 和 init。那么 app 的 index.js 是什么样子的呢？

```
export default function (kibana) {
    return new kibana.Plugin({
        require: ['elasticsearch'],

        uiExports: {
            app: {
                title: 'Indices',
                description: 'An awesome Kibana plugin',
                main: 'plugins/elasticsearch_status/app',
                icon: 'plugins/elasticsearch_status/icon.svg'
            }
        }
    });
}
```

这个示例中有两处特殊的代码：

1. `require` 指令加载了 elasticsearch 模块；这表示后续我们会用到这个模块，所以提前加载好。
2. `uiExports` 中使用了 app 键值对定义。其中这几对键值的含义如下：
    - title: app 的名称，用来显示在 Kibana 左侧边栏上的文字；

- icon: app 的图表，用来显示在 Kibana 左侧边栏上的图标；
- main: app 的主入口 js 文件。

服务器端部分

作为完整的 app，自然也还是有服务器端的部分。上节已经讲过，在 index.js 中的 init 部分定义这块：

```
return new kibana.Plugin({
      // ...
    init(server, options) {
        server.route({
            path: '/api/elasticsearch_status/index/{name}',
            method: 'GET',
            handler(req, reply) {
                server.plugins.elasticsearch.callWithRequest(req
, 'cluster.state', {
                    metric: 'metadata',
                    index: req.params.name
                }).then(function (response) {
                    reply(response.metadata.indices[req.params.n
ame]);
                });
            }
        });
    }
});
```

传入的 server 参数，我们在上节只是用来获取了一下 ESClient。这次，我们正式使用一些 Hapi.js 真正的功能。比如路由。

这里创建的是一个可以被 GET 方法访问的 URL 地址 `/api/elasticsearch_status/index/<index name>`。对访问的处理应答，使用 handler 方法。

handler 方法传入两个参数：

- req: 有关请求的信息都可以通过 req 对象获取，比如路由中捕获的 `<index name>` 就可以用 `req.params.name` 来引用。

- reply: 应答处理的内容通过 reply 返回。

然后采用 `server.plugins.elasticsearch.callWithRequest` 来发送 Elasticsearch 请求，通过 Promise.then 来异步返回最终的 reply。这部分和上节讲的类似，就不展开了。

前台界面的 app.js

index.js 和后台数据已经就绪，下面就是前台界面展示问题。我们已经在 index.js 里定义过了 main 文件，是 app.js。

The first two lines we will insert into the file are the following:

```
import 'ui/autoload/styles';
import './less/main.less';
import uiRoutes from 'ui/routes';
import uiModules from 'ui/modules';

import overviewTemplate from './templates/index.html';
import detailTemplate from './templates/detail.html';

uiRoutes.enable();
uiRoutes
.when('/', {
      template: overviewTemplate,
      controller: 'elasticsearchStatusController',
      controllerAs: 'ctrl'

})
.when('/index/:name', {
      template: detailTemplate,
      controller: 'elasticsearchDetailController',
      controllerAs: 'ctrl'

});
```

文件第一行永远要是加载 `ui/autoload/styles` 。这一行的作用是保证你的 app 界面和 Kibana 总体保持统一风格。这也是 Kibana5 才有的新内容。

然后通过 `uiRoutes` 来完成 Angular 框架的路由定义。这方面在之前的 Kibana 源码介绍中已经反复出现过。这里我们定义好路由对应的控制器和模板文件。

控制器

作为一个简单示例，我们可以直接在 app.js 里继续实现控制器部分。如果是复杂应用，一般这里可以拆分成单独文件。

在 Kibana 中，实现控制器的方式如下：

```
uiModules
.get('app/elasticsearch_status')
.controller('elasticsearchStatusController', function ($http) {
    $http.get('../api/elasticsearch_status/indices').then((respo
nse) => {
        this.indices = response.data;
    });
});
```

这里采用的是 Kibana 框架已经封装好的 `uiModules.get().controller()`，比标准的 `angular.module` 省去了一些创建声明、依赖处理之类的工作。同样也是之前的源码讲解里很熟悉的部分了。

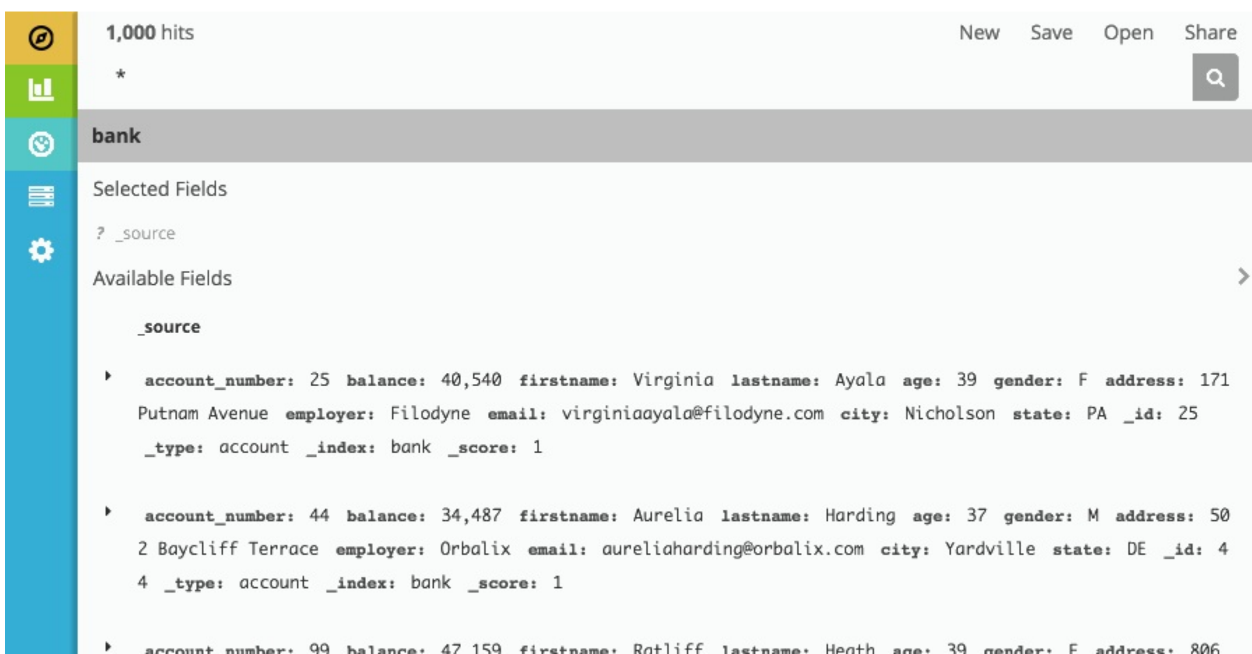
这里作为和后端的配合，我们使用 Angular 标准的 `$http` 来调用 `../api/elasticsearch_status/indices` 地址。这正是之前我们声明好的服务器端 URL。

页面模板

Angular 模板语言也是已经见过很多面的老朋友了，这块我们用最简单的一个 `ng-repeat` 循环展示列表即可。

```
<div class="container">
  <div class="row">
    <div class="col-12-sm">
      <h1>Elasticsearch Status</h1>
      <ul class="indexList">
        <li ng-repeat="index in ctrl.indices">
          <a href="#/index/{{index}}">{{ index }}</a>
        </li>
      </ul>
    </div>
  </div>
</div>
```

完毕。一个带有前后台乃至菜单栏的完整 app 就是这么简单：



Kibana 截图报表

Elastic Stack 本身作为一个实时数据检索聚合的系统，在定期报表方面，是有一定劣势的。因为基本上不可能把源数据长期保存在 Elasticsearch 集群中。即便保存了，为了一些已经成形的数据，再全面查询一次过久的冷数据，也是有额外消耗的。那么，对这种报表数据的需求，如何处理？其实很简单，把整个 Kibana 页面截图下来即可。

FireFox 有插件用来截全网页图。不过如果作为定期的工作，这么搞还是比较麻烦的，需要脚本化下来。这时候就可以用上 phantomjs 软件了。phantomjs 是一个基于 webkit 引擎做的 js 脚本库。可以通过 js 程序操作 webkit 浏览器引擎，实现各种浏览器功能。

phantomjs 在 Linux 平台上没有二进制分发包，所以必须源代码编译：

```
# yum -y install gcc gcc-c++ make flex bison gperf ruby \
  openssl-devel freetype-devel fontconfig-devel libicu-devel sql
ite-devel \
  libpng-devel libjpeg-devel
# git clone git://github.com/ariya/phantomjs.git
# cd phantomjs
# git checkout 2.0
# ./build.sh
```

想要给 kibana 页面截图，几行代码就够了。

Kibana3 截屏代码

下面是一个 Kibana3 的截屏示例。 `capture-kibana.js` 如下：

```
var page = require('webpage').create();
var address = 'http://kibana.example.com/#/dashboard/elasticsear
ch/h5_view';
var output = 'kibana.png';
page.viewportSize = { width: 1366, height: 600 };
page.open(address, function (status) {
    if (status !== 'success') {
        console.log('Unable to load the address!');
        phantom.exit();
    } else {
        window.setTimeout(function () {
            page.render(output);
            phantom.exit();
        }, 30000);
    }
});
```

然后运行 `phantomjs capture-kibana.js` 命令，就能得到截图生成的 kibana.png 图片了。

这里两个要点：

1. 要设置 `viewportSize` 里的宽度，否则效果会变成单个 panel 依次往下排列。
2. 要设置 `setTimeout`，否则在获取完 index.html 后就直接返回了，只能看到一个大白板。用 phantomjs 截取 angularjs 这类单页 MVC 框架应用时一定要设置这个。

Kibana4 的截屏开源项目

Kibana4 的 dashboard 同样可以采取上面 Kibana3 的方式自己实现。不过社区最近有人完成了一个辅助项目，把截屏配置、历史记录展示等功能都界面化了。项目地址见：https://github.com/parvez/snapshot_for_kibana。



Elastic Stack 与 Hadoop 体系的区别

Kibana 因其丰富的图表类型和漂亮的前端界面，被很多人理解成一个统计工具。而我个人认为，ELK 这一套体系，不应该和 Hadoop 体系同质化。定期的离线报表，不是 Elasticsearch 专长所在(多花费分词、打分这些步骤在高负载压力环境上太奢侈了)，也不应该由 Kibana 来完成(每次刷新都是重新计算)。Kibana 的使用场景，应该集中在两方面：

- 实时监控

  通过 histogram 面板，配合不同条件的多个 queries 可以对一个事件走很多个维度组合出不同的时间序列走势。时间序列数据是最常见的监控报警了。

- 问题分析

  通过 Kibana 的交互式界面可以很快的将异常时间或者事件范围缩小到秒级别或者个位数。期望一个完美的系统可以给你自动找到问题原因并且解决是不现实的，能够让你三两下就从 TB 级的数据里看到关键数据以便做出判断就很棒了。这时候，一些非 histogram 的其他面板还可能会体现出你意想不到的价值。全局状态下看似很普通的结果，可能在你锁定某个范围的时候发生剧烈的反方向的变化，这时候你就能从这个维度去重点排查。而表格面板则最直观的显示出你最关心的字段，加上排序等功能。入库前字段切分好，对于排错分析真的至关重要。

Splunk 场景参考

关于 elk 的用途，我想还可以参照其对应的商业产品 splunk 的场景：

> 使用 Splunk 的意义在于使信息收集和处理智能化。而其操作智能化表现在：
>
> 1. 搜索，通过下钻数据排查问题，通过分析根本原因来解决问题；
> 2. 实时可见性，可以将对系统的检测和警报结合在一起，便> 于跟踪 SLA 和性能问题；
> 3. 历史分析，可以从中找出趋势和历史模式，行为基线和阈值，生成一致性报告。

```
-- Peter Zadrozny, Raghu Kodali 著/唐宏，陈健译《Splunk大数据分析》
```

推荐阅读

- Elasticsearch 权威指南
- 精通 Elasticsearch
- The Logstash Book
- Elastic Stack Advent

致谢

- 感谢 crazw 完成 collectd 插件介绍章节。
- 感谢 松涛 完成 nxlog 场景介绍章节。
- 感谢 LeiTu 完成 logstash-forwarder 介绍章节。
- 感谢 jingbli 完成 kafka 插件介绍章节。
- 感谢 林鹏 完成 ossec 场景介绍章节。
- 感谢 cameluo 完成 shield 介绍章节。
- 感谢 tuxknight 指出 hdfs 章节笔误。
- 感谢 lemontree 完成 marvel 介绍章节。
- 感谢 NERO 完成 utmp 插件开发介绍章节。
- 感谢 childe 完成 kibana-auth-CAS 介绍, elasticsearch 读写分离和别名等章节。
- 感谢 gnuhpc 完善 elasticsearch 插件 index 配置细节，logstash-fowarder-java 部署细节。
- 感谢 东南 指出两处代码示例笔误。
- 感谢 陆一鸣 指出 openjdk 的 rpm 名错误。
- 感谢 elain 完成 search-guard 章节。
- 感谢 wdh 完成 search-guard v2 版章节。
- 感谢 李宏旭 完成 hdfs 快照备份 章节。
- 感谢 abcfy2 完成 grafana 章节。

捐赠者名单

感谢以下童鞋的捐助，让我有动力持续下来并且注册了 http://kibana.logstash.es 这么个有意思的域名用来发布本书：

donor	value
颜*	50.00
赵*远	10.00
韩*光	22.00
刘*卫	10.00
185**6322	5.00
彭*源	6.11
余*	100.00
李*灿	20.00
孙*	101.00
彭*根	50.00
史*春	10.00
张*	18.76
陈*林	66.00
吴*维	10.00
周*	10.00
叶*	50.00
张*贝	51.00
刘*	10.00
肖*宇	25.60
周*	100.00
高*达	10.00
庞*	30.00
林*光	10.00

张*	20.00
刘*	10.00
周*文	131.21
俞*	24.80
宋*	4.01
吴*洪	15.00
王*	28.00
金*	110.00
段*	6.66
叶*荣	55.00
闫*苗	100.00
张*梁	20.00
董*	5.50
庄*宇	5.00
李*平	1000.0
wiki奇	10.00
没谱的曲	10.00
陈*	20.00
弓长小宝	147.52
彦达	20.00
剑	20.00
志超	100.00
焕良	10.24
allen	30.00
紧-^ ^	10.00
正翔	20.00
戒小贤	100.00
纳音妙德	29.99
王先生	5.00
睿	16.60

勇勇	30.00
HugoYang	100.00
春林	30.00
承斌	10.00
彦峰	30.00
冥炎	3.00
毓	5.00
小蜜蜂	20.00
锐	50.00
岛思族	38.00
奕庆	20.00
公孙	5.00
NoBody™	6.66
王*勇	100.00
moonwolf	18.88
JoeZhu	70.00
云雾中的月光	10.00
Sandy	20.00
沈宽	20.00
萧田国	40.00
吕建飞	50.00
likuku	10.00
胖子黄	10.00
肖力	10.00
brad	10.00
ming	10.00
ATOM	10.00
TonyWu	10.00
言岳	10.00
海盗旗	20.00

丁天密	20.00
Aroline	50.00
李学明	20.00
余弦	50.00
卫向	10.00
shuanggui	10.00
Jacob.Yu	10.00
陈洋	10.00
吴晓刚	10.00
肖平Jacky	10.00
马亮	10.00
袁乐天	10.00
魏振华	100.00
刘宇	50.00
吕召刚	20.00
Bin	10.00
华仔	20.00
山野白丁	20.00
陈自欣	10.00
我叫于小炳	20.00
军呐	13.00
吴顺超	50.00
辉	10.00
kimura	20.00
王小力	20.00
猜猜	5.00
恩华	5.00
森	5.00
刚刚	15.00

泽灿	10.00
邓*毅	5.00
小彬	10.00
小罗罗	19.00
哈里	5.00
小史sk2	10.00
如水东流	50.00
蜗牛	50.00
义芳	5.00
守望海豚	15.00
JEFF	10.00
郑方方	10.00
Daniel	20.00
Duandy	50.00
赵班长	100.00
dyson	42.00
敬辰	0.01
NEXON	8.88
亚峰	1.00
强	12.99
波罗学	3.00
段红翎	10.00
奇弦	5.00
大鹏	10.00
祥帅	5.00
文龙	10.00
mkods	25.00
建华	2.00
纪念	22.22
张威	5.00

sala	10.00
博文	10.00
Rocks	20.00
剑平	10.00
昊	3.00
清和	60.00
兜兜有糖	2.55
蕾	50.00
小疆	10.00
俊伟	10.00
花开时节	4.50

共计：4970.69 元，域名注册已用 275.50 元(3 年)，QQ群升级已用 628.00 元(2 年)。